A GENERAL PARSING ALGORITHM FOR CONTEXT-FREE GRAMMARS

A Thesis

Presented to

the Faculty of Graduate Studies and Research

The University of Manitoba

In Partial Fulfilment

of the requirements for the Degree

Master of Science

in the Institute for Computer Studies

---

by

Rainer Kossmann

February 1971

ABSTRACT

The work presented in this thesis arose out of an investigation concerning the compilation of a new programming language, namely Algol-68. A radically new syntax representation used in the definition of this language makes the syntax analysis of this language a major problem. One promising method of handling this problem is the transformation of most of the Algol-68 syntax to a context-free grammar which is readily handled by well-known and existing methods. A brief description of this grammar transformation is given in chapter 5 of this thesis.

A further result of the preliminary investigation was that it was found to be extremely useful and possibly necessary to have available a general parsing algorithm for context-free grammars. This algorithm was developed by the writer and forms the major subject of this thesis. The theory of this algorithm is presented in chapters 1 through 4. A listing for a Pl/1 program, fully working but still under development, plus a few examples are appended.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. P. R. King, for the help and encouragement that he provided throughout the development of this work.

Further thanks go to the Department of Computing Science which made available the text-editing facilities used in the preparation of this thesis. This feature has saved a considerable amount of work which would have been required in the preparation of multiple drafts by manual means.

Finally, I would like to thank my wife, Maureen, for her patience with a somewhat temperamental machine and for many hours spent in typing the original draft of this thesis into the computer files.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

Many parsing algorithms for context-free grammars require the grammar to be restricted to some proper subset of context-free grammars. Indeed, general parsing algorithms for context-free grammars are rather scarce. This is not by accident since a general parsing algorithm tends to be of a rather inefficient nature. By placing a few restrictions on the type of context-free grammar that a parsing algorithm will work for, it is often possible to derive greatly increased efficiency in the parsing algorithm for such grammars.

One example which demonstrates this clearly consists of the Wirth-Weber precedence grammars (1). Efficient parsing algorithms for these grammars exist

and a recent paper by A. Learner and A. L. Lim demonstrates an algorithm that will transform any context-free grammar into an equivalent Wirth-Weber precedence grammar (2).

A similar state of affairs exists for other classes of context-free grammars which are subject to some restriction. It is often possible to find a grammar satisfying a given set of restrictions but which is at the same time equivalent to a given context-free grammar.

The obvious question that one must ask now is whether it is worthwhile to design a general parsing algorithm for context-free grammars, even though such parsing algorithms may prove too inefficient to be feasible as a parsing algorithm for a production compiler, or interpreter.

The answer is an emphatic "yes" for the following reason:

The language designer is not as concerned

initially with the type of grammar he must provide as with the type of problem the grammar must solve. The design of the grammar is thus his major problem and will require the bulk of his effort. He should thus be expected to have at his disposal the largest possible set of grammars. Furthermore, he should expect to do a minimal amount of rewriting of a grammar he has designed in order to test the grammar under some system. A general parsing algorithm for context-free grammars fulfills these requirements very nicely as the language designer is provided with a tool that will parse any context-free grammar. Once a grammar for a particular application has been designed, the language designer may then of course consider the problem of transforming his grammar into one which will conform to the restrictions imposed by a particular parsing algorithm which he wishes to use in a production compiler or interpreter.

The reason just cited, and the fact that no

general top-down, left-right parsing algorithms for context-free grammars were known to the writer, were considered sufficient justification for the development of such an algorithm.

The remainder of this chapter will be dedicated to providing the basic framework of definitions within which the thesis will be presented.

1.1 BASIC DEFINITIONS

A convention adopted throughout the thesis is that single, underlined, capital letters represent sets whereas single capital letters, indexed or unindexed, represent single elements of a set. Thus, "$\underline{B}$" represents a set and both "B" and "B_1" represent single elements of some set. Additionally the notation "$\underline{A}$*" is meant to represent the set of all strings that can be formed with elements of the set $\underline{A}$.

The syntax of a context-free phrase structure grammar is expressed with the aid of a finite alphabet $\underline{A}$. in this thesis $\underline{A} = \underline{S} \cup \underline{D}$ where

$\underline{S}$ = {a, b, c, ..., x, y, z} is the set of "syntactic marks", except where otherwise stated

$\underline{D}$ = {:, ;, ., ,} is the set of "other syntactic marks".

Define $\underline{S}^*$ as the set of (nonempty) strings over $\underline{S}$. The elements of $\underline{S}^*$ are termed "protonotions." Define a context-free phrase structure grammar (c.f.p.s.g.) as a 4-tuple

($\underline{V}$, $\underline{R}$, symbol, Z) where

$\underline{V} \subset \underline{S}^*$ is a finite set, known as the set of "notions",

$Z \in \underline{V}$ is a particular notion known as the "head",

$\underline{R}$ is a finite set of rules of the form

1) A: . called the empty rule, or

2) $A: B_1, B_2, \ldots, B_n.\ n \geq 1$

where $B_i \in \underline{V}$, $i=1,2,\ldots,n$ and $A \in \underline{N}$ where $\underline{N} = \underline{V} - \underline{T}$. $\underline{T}$ is the subset of $\underline{V}$ consisting of those elements of $\underline{V}$ which end with the particular protonotion "symbol". Elements of $\underline{T}$ are called (terminal) symbols. Elements of $\underline{N}$ are called non-terminals.

Additionally, any set of rules of $\underline{R}$ with identical left hand sides, say "A: B., A:C., ..., A:Z." are rewritten as "A: B; C; ...; Z.". Furthermore, there must not be any rules $A: B. \in \underline{R}$ such that $A \in \underline{T}$.

A few further definitions are required. Let members of the set $\{u,v,w,x,y,z\}$ be elements of $\underline{V}^*$. We say that "w directly produces v (written '$w \Longrightarrow v$') by application of the rule U:u." if there are (possibly empty) strings "x" and "y" such that "w=xUy" and "v=xuy".

We say that "w produces v" (written $w \overset{*}{\Longrightarrow} v$) if

1. $w = v$ or

2. $w \overset{*}{\Longrightarrow} u$ and $u \Longrightarrow v$, where $u \in \underline{V}^*$.

The "language generated by a grammar G" (written L(G)) is defined as

$L(G) = \{z \mid Z \overset{*}{\Longrightarrow} z \text{ and } z \in \underline{T}^*\}$ where, as before, "Z" is the head of the grammar G and "$\underline{T}$" is the set of "symbols" of the grammar G.

To explain these definitions a little more clearly, a short example is now in order. Consider the context-free, phrase structure grammar defined by the 4-tuple ($\underline{V}$, $\underline{R}$, symbol, Z) where

$\underline{V} = \{$identifier, letter, digit, asymbol, bsymbol, onesymbol$\}$,

$\underline{R}$ is the set of rules

{identifier: letter; identifier, letter; identifier, digit.

letter: asymbol; bsymbol.

digit: onesymbol.}

"symbol" is the protonotion "symbol", and Z is the notion "identifier".

Then the protonotions such as "identifier", "letter", and "asymbol" are also notions by virtue of the fact that they are in V.

The set of rules R consists of three rules. To aid in the reading of these rules, the other syntactic marks {:, ;, ., ,} may be read as follows:

"is defined to be" for ":",

"or" for ";", and

"followed by" for ",".

The other syntactic mark "." is not read but serves merely to indicate the end of a rule.

Thus, the rules in R may now be read as

"(an) identifier is defined to be (a) letter or (an) identifier followed by (a) letter or (an) identifier followed by (a) digit."

"(a) letter is defined to be (an) asymbol or (a) bsymbol."

"(a) digit is defined to be (a) onesymbol."

Quantities in brackets ( ) have been added to improve readability.

It should now be noted that the members of T are not further defined by R. The implication is that the definition of symbols is to be provided by the implementor of a language defined by the afore mentioned syntax. The implementor is thus free to choose his own representation for a symbol, be it an English alphabetic character, a Greek one, or possibly several different characters for the same symbol.

The symbols may thus be regarded as undefined notions and it may be considered that an implementor of a language will provide his own definitions such as for example:

asymbol : a.

asymbol : A.,

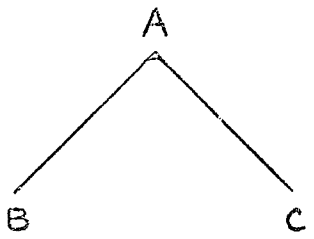
or else will provide a routine that will be used in recognizing symbols.

Members of T may thus be regarded just as any other notions and their use effectively removes from the syntax definitions any terminal symbols.

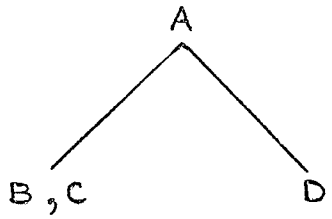
1.2 THE TREE STRUCTURE FORM OF SYNTAX DEFINITIONS

Occasionally, a particular example will be more easily understood if the syntax definitions are represented in the form of a tree structure. The convention that will be adopted here is that the

lefthand side of a rule will be a node of the tree and there will be one branch emanating downward from this node for each alternative of the rule. Thus, the rule A : B; C. would have the following appearance in a tree structure:



If an alternative consists of more than one notion, then these are represented on the tree as nodes separated by a comma. Thus, A : B, C; D. would have the appearance



In the latter case, there will be a set of branches emanating from each such node of an alternative with multiple nodes. Thus, the set of rules

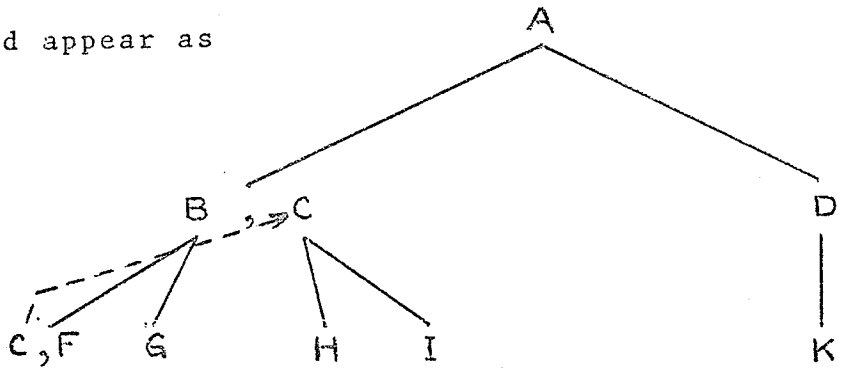
A : B, C; D.

B : C, F; G.

C : H; I.

D : K.

would appear as



It may now happen that a node such as C in the rule B : C, F; G. has been previously defined in the tree. In that case, it will not be necessary to generate a new subtree for this node C. Instead, a dotted line to the previously defined C will indicate where the definition for node C may be found in the tree.

If the syntax tree for a context-free, phrase structure grammar is drawn up, there will be two types of nodes in the tree:

a) those which have branches emanating downward from the node

b) and those which do not have branches emanating downward from them.

The nodes that do not have downward emanating branches may be called end nodes. There exist two types of end nodes:

a) those that refer back into the tree because a previous definition for the node exists

b) those nodes which are symbols and for which no definition exists.

To conclude the discussion on the definition tree for context-free grammars, one important point must be stressed. This point is the distinction that exists between notions and nodes. A "notion" is a member of $\underline{V}$ which may however occur as more than one "node" of the definition tree for a context-free grammar. These "nodes" are quite distinct from each

other.

In further discussions, the concept of "node" will be extended to cover the "node of a syntax rule". Thus, in the syntax rule

A: B, C; B, D.,

the "notions" A, B, C, and D are "nodes" within that syntax rule. Note further that the first node "B" of the syntax rule is quite distinct from the second node "B" of that syntax rule. It is evident that, if a definition tree is drawn up for a syntax rule, each node of the syntax rule will be represented as a node on the definition tree and to each node of the definition there corresponds a node of the syntax rule.

The basic definitions within which the thesis will be presented have now been covered. The following chapter will be devoted to presenting a parsing algorithm due to Floyd which will form the major part of the general algorithm presented in this thesis.

CHAPTER 2

FLOYD'S ALGORITHM

The parsing algorithm presented in this chapter is a modified version of an algorithm due to Floyd (3). It is a top-down, left-right parsing algorithm which includes backtracking.

In this chapter, a short section on terminology will be followed by a description of the two major data areas used by the parsing algorithm; the syntax table and the analysis record. Following this, the operation of the parsing algorithm will be discussed along with an example that demonstrates its operation.

2.1 THE SYNTAX TABLE

In this section, the structure of the syntax table will be explained. However, some of the terminology used must first be defined.

2.1.1 TERMINOLOGY

Suppose the following is a rule of a context-free grammar.

A:B;C,D.

Then B is called a "son" of A, A is called a "father" of B, and D is called the "next component" or simply "component" of C. Infrequently, C may be referred to as the "previous component" of D. Note that A is also the father of C and D and that C and D are sons of A. C is further called the "brother" of B.

2.1.2 THE TABLE

The approach used here is for the syntax definitions to be stored in a list structure form instead of a string of syntax definitions as used by Floyd (3). The advantage of using a list structure derives from the fact that fast parsing algorithms may be constructed for grammars specified in this form, especially if the parsing algorithm is written in a low

level language such as Assembler. At the same time, the parsing algorithm remains simple for grammars specified in list structure form (see also 8).

The list structure form of the syntax definitions as used here consists of one four element line in a syntax table for each definition. This line has the following configuration:

brother	component	son	code

The fields for the brother, component, and the son contain pointers to the respective rule within the syntax table for these quantities. For example, in the rule A: B; C, D., there will be a line in the syntax table for the notion A. The "son" field for the notion A will contain a pointer which points to the line in the syntax table in which the definition for B is given.

However, there is a brother to the notion B and

this brother consists of the notion C followed by the notion D. To note this fact, there will be a pointer in the "brother" field of B that points to the line in the syntax table in which the notion C is defined. The "component" field of the notion C will contain a pointer to the line in which the notion D is defined.

Absence of a brother or a component is noted by a zero in the corresponding field.

As yet, no reason has been given for the existence of the code field and this situation will now be rectified. It was mentioned previously that no terminals exist within the syntax definitions but there are "symbols" for which no further definitions exist. The parser must be able to recognize the occurrence of symbols and this is done by means of placing a terminal symbol code in the code field of a notion which is a symbol.

Recognition of terminals will be done by a recognizer routine which is activated whenever a code field contains a terminal symbol code. The subordinate

field of the line in the syntax table for a symbol will contain a pointer which points into a terminal symbol list and which will be used by the recognizer in trying to recognize a symbol.

Employing the scheme of a terminal symbol list for the recognition of terminals, it is relatively easy to construct a recognizer which not only recognizes single characters but also syntactic units which consist of the concatenation of two or more characters or which have several different character representations.

The parser must now test the code of every notion considered during a parse to determine whether or not the notion is a symbol. Great gains in generality can now be realized by making the code field a pointer which will point to a routine; the recognizer in the case of symbols.

As with many other topics, theory and practice do not always go hand in hand in the parsing of grammars. The difficulty in parsing of grammars of

course is that it is possible to construct context-free grammars in theory but that in practice the programming languages are generally based on syntax definitions which are not entirely context-free. It is thus necessary to be able to carry out special actions for some notions other than symbols. This is facilitated in the approach used above at no extra cost to the parsing algorithm by simply placing pointers to a special handling routine into the code field of notions requiring such special handling. The need for recognizing symbols of course is the reason why other special handling routines can be incorporated at no extra cost to the parsing algorithm.

As an example, the following grammar previously used in Chapter 1 will be encoded into a syntax table. Let R be the set of rules

{identifier: letter; identifier, letter;
identifier, digit.
letter: asymbol; bsymbol.
digit: onesymbol.}

Then, the following table is the syntax table for this grammar:

SYNTAX TABLE

#	Notion	Broth	Comp	Son	Code
1.	identifier.	0	0	2	0
2.	letter;	3	0	7	0
3.	identifier,	5	4	2	0
4.	letter;	5	0	7	0
5.	identifier,	0	6	2	0
6.	digit.	0	0	9	0
7.	asymbol;	8	0	1	t
8.	bsymbol.	0	0	2	t
9.	onesymbol.	0	0	3	t

Thus, line 1 states that the son for "identifier" is to be found in line 2; i.e. the syntax definition for "identifier" begins in line 2. Line 2 states that the "identifier" of line 1 may be possibly a letter which is defined in line 7 (pointed at by "son") or alternatively may be defined in line 3 (pointed at by "brother").

It may therefore be said that an alternative definition for "identifier" is to be found in line 3. This alternative definition consists of the two components "identifier, letter".

The "4" in the "component" field of line 3 points out the fact that the fourth line, a definition for letter, is the "next component" of line 3. Thus, lines 3 and 4 are just one of the definitions of the "identifier" of line 1.

Of interest now is the "son" field of line 3. Since line 3 must define an "identifier", the fact that "identifier" has been previously defined in line 1 is noted. In line 1, one finds that "identifier" is defined starting at line 2 (see the "son" field of line 1) and thus, this 2 is merely copied in to the "son" field of line 3. Line 3 therefore states that "a definition for identifier may be found starting at line 2".

Only the "symbols" of lines 7,8, and 9 remain unexplained. The "t" in the code field of these lines

is a code which indicates that these are terminals and require a recognizer. If any of these lines is made a goal of the parsing algorithm, the recognizer will use the "son" of these lines as an indication of what "symbol" must be recognized. For example, in this case, the 1, 2, or 3 might indicate the first, second, or third symbol in some terminal symbol list which is used by the recognizer.

2.2 THE ANALYSIS RECORD

The analysis record is an area in which details of a particular parse are recorded. This analysis record will normally be stored and passed to a semantic routine at a later stage for the purpose of code generation. Before going into details about the structure of the analysis record, it is again helpful to first define a few terms.

2.2.1 TERMINOLOGY

Each line in the analysis record will be of form

pointer	superior	subordinate	predecessor

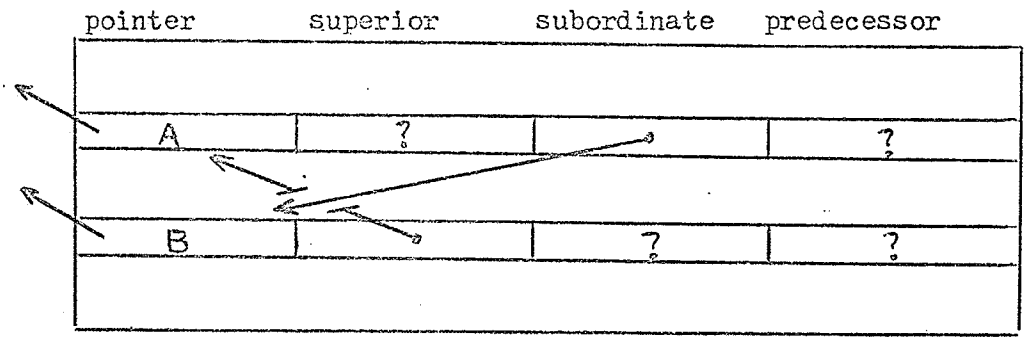
and will have a line number denoting its position within the analysis record.

At any stage during the parse, the parsing algorithm will be considering a goal and there will be one line in the analysis record corresponding to this goal. "Pointer" in the above analysis record serves as a link to the syntax definitions which ties the line in the analysis record to the specific goal in the syntax tables. This goal under consideration will in general have a father, a son, and a previous component and there will generally be lines in the analysis record for each of these quantities. In the above diagram, superior, subordinate, and predecessor are links which hold the line number within the analysis record of an

output line for the father, son, and the previous component respectively. Absence of one of these quantities is denoted by a zero in the respective field. The terms "pointer", "superior", "subordinate", and "predecessor" have now been defined and the operations carried out with the analysis record are discussed next.

2.2.2 THE ANALYSIS RECORD ORGANIZATION

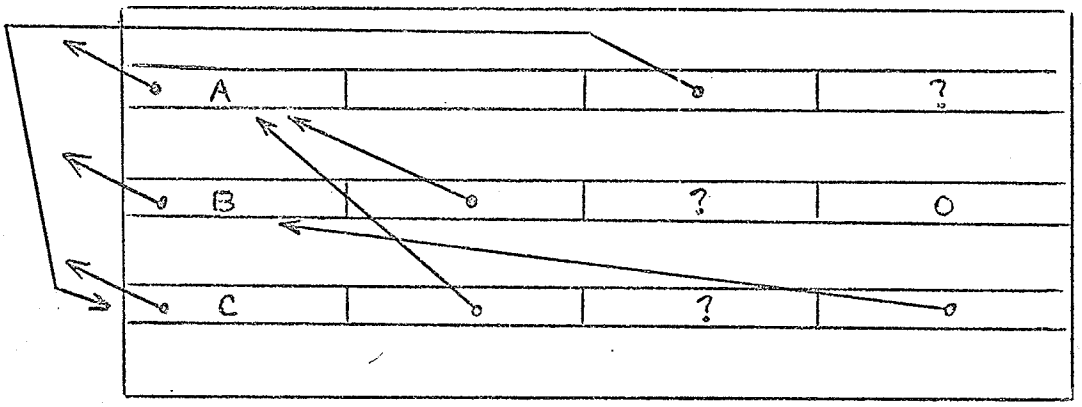
Since a goal may have more than one subordinate, the subordinate in the analysis record will contain a pointer to the last subordinate considered. Consider for example the rule A: B, C. During the parse, a line will exist in the analysis record for the goal A. When the goal B is considered as a subgoal, the subordinate field of the line for A will contain a pointer to the line for goal B as shown below:



Once B has been recognized, C must be made the next subordinate of A. This is done by generating a line in the analysis record for C with the following rearrangement of pointers:

1) the subordinate of A is placed in the predecessor field of C

2) a pointer pointing to the new line for C is placed in the subordinate field of A.



Thus, it is possible to reconstitute an analysis by locating the last subordinate of a goal as indicated by the subordinate pointer for that goal and then chaining back through the predecessors to locate all

subordinates of the goal.

The following analysis record constitutes the analysis of the identifier "ab1". The grammar used is the same grammar defining "identifier" that was used previously. The syntax table for this grammar was previously given on page 20.

ANALYSIS RECORD

#	type	point	sup	sub	pred
1	identifier	1	0	8	0
2	identifier	5	1	6	0
3	identifier	3	2	4	0
4	letter	4	3	5	0
5	asymbol	7	4	0	0
6	letter	4	2	7	3
7	bsymbol	8	6	0	0
8	digit	6	1	9	2
9	onesymbol	9	8	0	0

The reader may have noticed in the discussion of the syntax table and the analysis record that the quantities "father" and "son" of the syntax table appear to be identical to the quantities "superior" and

"subordinate" of the analysis record. This is not quite correct. The quantities "father" and "son" are links which connect the syntax definitions within the syntax table. Quantities "superior", "subordinate", and "predecessor" on the other hand are links within the analysis record which connect the various components of an analysis. To emphasize this distinction between these two sets of pointers and to avoid possible confusion between them, they have purposely been given different names. Thus, whenever reference is made to the "father", "son", "previous component", or "next component", it is immediately obvious that a line of the syntax tables is intended. Reference to "superior", "subordinate", or "predecessor" on the other hand immediately makes it known that a line of the analysis record is being referred to.

The connection between these two tables is of course via the "pointer" of the analysis record. It indicates the line of the syntax table to which the line of the analysis record corresponds.

2.3 THE PARSING ALGORITHM

To explain the logic of the parsing algorithm, a metaphor due to Floyd is reproduced here(3). Only the statement of syntactic rules has been changed to conform to the convention adopted in this thesis.

"Suppose a man is assigned the goal of analyzing a sentence in a phrase structure language of known grammar. He has the power to hire subordinates, assign them tasks, and fire them if they fail; they in turn have the same power. The convention will be adhered to that each man will be told only once "try to find a G" where G is a notion of the language, and may thereafter be repeatedly told 'try again' if the particular instance of a G which he finds proves unsatisfactory to his superiors. Depending on the form of the definition of G, each subordinate (e.g., S) should adopt an appropriate strategy:

1. If G is a terminal character, and if it is the next character of the sentence, S must cover

the character, and report success to his superior. If it is not the next character of the sentence, S must report failure. After success, if told by his superior to try again, S must report failure and uncover the character.

2. If G:G1., S must appoint a subordinate S1 with the command, 'try to find a G1'. S repeats S1's report to his supperior, firing S1 on a report of failure. If told to try again, S must tell S1 to try again, again transmitting the report to his superior and firing S1 on failure.

3. If G:G1, G2, ..., Gn., S must appoint successively one subordinate Si for each Gi, with the command, 'try to find a Gi.' If Si succeeds, i is increased by one, a new subordinate hired and the process repeated until i is greater than n, when S reports success. If Si fails Si is fired, i is decreased by one and if i is greater than zero,

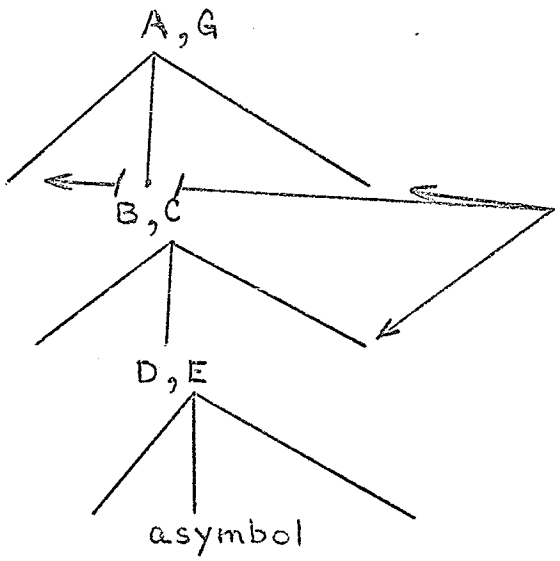
the new Si (predecessor of he who failed) told to try again. If i = 0, S reports failure, having exhausted all ways of finding a G. If after success, S is told to try again, he sets i = n, tells Si to try again, and proceeds as before on Si's report.

4. If G:G1;G2; ... Gn., S must appoint successively one subordinate Si for each Gi, with the command, ' try to find a Gi.' If S fails he is fired, i is increased by one, a new subordinate hired, and the process repeated until i is greater than n, when S reports failure. If Si succeeds, S reports success. If after success S is told to try again, he tells Si (who succeeded) to try again, and proceeds as before on Si's report.

5. All more complicated definitions can be regarded as built up from the first four types."

It is evident that the backtracking feature of

the algorithm is the 'try again' facility. Since the operation of this feature may be difficult to grasp, it is advantageous to visualize this operation diagramatically. Suppose a goal A which has a successor G has the following definition tree associated with it:



Then, supposing that success had been reported for A via the notions lying along the path from A to asymbol, it may be that a G can not be found. In that case, A is told to try again and the parser will proceed downward along the previously recognized branches of the tree with try again commands. This downward procession along the branches of the tree is of course

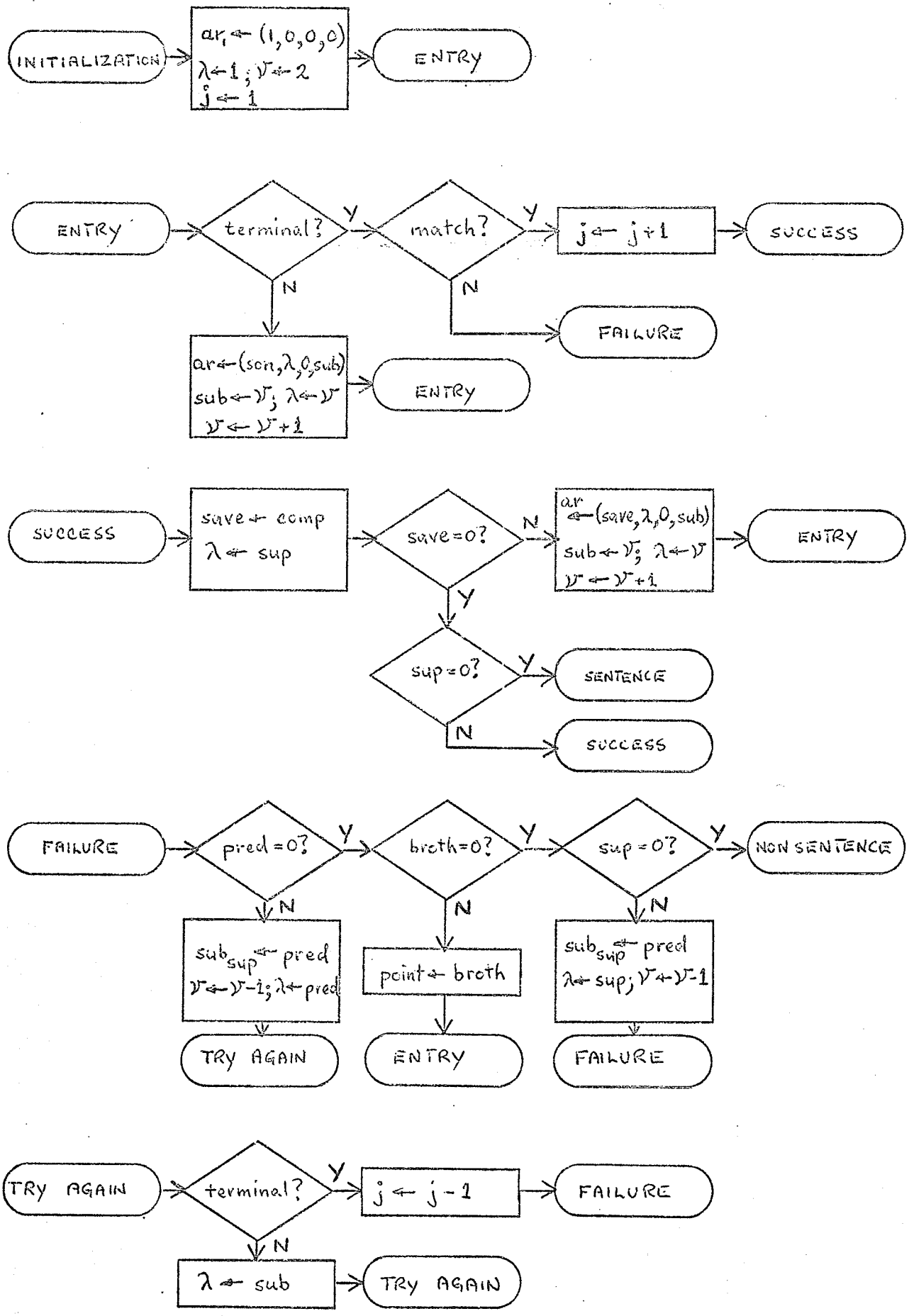
possible by chaining through the subordinates stored in the analysis record.

As soon as this process leads to the lastmost recognized terminal, i.e., asymbol in this case, the string pointer is reset so as to uncover asymbol and failure is reported to asymbol's superior (notion E in this case). At this point, the parsing algorithm exits from the try again mode and continues parsing normally (in other words, the next alternative to asymbol is tried in an attempt to satisfy goal E etc.).

Thus, using the try again feature, the entire subtree defining A is scanned before failure is reported for A. (This contrasts to normal parsing algorithms which, after having met with success for the goal A, will try the next alternative of "A, G." if G should happen to fail. Thus, normal parsing algorithms may leave part of the definition tree for A unexamined.)

The complete parsing algorithm is presented on the next page as a flowchart. A short description of

FLOYD'S ALGORITHM



the terminology used in the flowchart follows:

1. "ar " represents the i^{th} line in the analysis record. It is implicitly subscripted by γ if no subscript is given.

2. "$point_i$", "sup_i", "sub_i", and "$pred_i$" represent respectively the pointer, superior, subordinate, and predecessor fields of the i^{th} line of the analysis record. These quantities are always implicitly subscripted by λ if no subscript is given.

3. "λ" indicates the current line in the analysis record on which the parsing algorithm is operating.

4. "γ" indicates the first empty line in the analysis record.

5. "j" indicates the location of the next, as yet unrecognized character in the input string.

6. "broth", "comp", and "son" are respectively the brother, next component, and son of the syntactic goal currently under consideration. Note that the syntactic goal currently under consideration is recorded in $point_\lambda$. Thus, "broth", "comp", and "son" may be obtained by using the link stored in $point_\lambda$ to index the syntax table. It should also be noted that access to the code field of the syntax table is possible in the same manner. This code field must of course be accessible in order that a decision as to whether or not the goal is a terminal can be made.

7. "save" is a temporary save area.

The meaning of the flowchart statement "$sub_{sup} \leftarrow pred$" will thus be:

1. Determine the contents of sup_λ and use this as an index "k" to sub.

2. Carry out the operation $sub_k \longleftarrow pred_\lambda$; i.e. move the contents of $pred_\lambda$ to sub_k.

The meaning of the flowchart should now be self-explanatory.

To conclude this chapter, a complete example demonstrating how a typical grammar is handled and parsed will be presented. Let G = ($\underline{V}$, $\underline{R}$, symbol, Z) be a c.f.p.s.g. where

$\underline{V}$ = {expression, identifier, asymbol, bsymbol, plussymbol},

Z= expression,

$\underline{R}$ is the following set of rules:

{expression: identifier, plus symbol, bsymbol.

identifier: asymbol; identifier, asymbol; identifier, plussymbol.}

Assuming the asymbol is represented by "a", the bsymbol by "b", and the plussymbol by "+", then clearly

$$L(G) = \{as+^{q}b \mid q \geq 1,\ s \in \{a,+\}^{*}\}.$$

However, for $q \geq 1$, the first q-1 plussymbols are part of the identifier whereas the last plussymbol is an operator.

Clearly, there is some local ambiguity here even though the grammar on the whole is unambiguous. Consider the problem of the top-down parsing algorithm which has as its subgoal the "identifier" of "expression". It must decide how many of the plussymbols after the initial "as" are part of the identifier. (Note that it is not correct in this case to search for the longest identifier.) The backtracking feature of Floyd's algorithm allows this local ambiguity to be resolved because, if an identifier has been recognized incorrectly, i.e. the identifier is not of the right length, then the algorithm will "try again" at a later stage to find another identifier until all possibilities have been exhausted.

The following analysis of the expression "a++b"

is provided as an illustration.

SYNTAX TABLE

#	Type	Broth	Comp	Son	Code
1	expression.	0	0	2	0
2	identifier,	0	3	5	0
3	plussymbol.	0	4	3	t
4	bsymbol.	0	0	2	t
5	asymbol;	6	0	1	t
6	identifier,	8	7	5	0
7	plussymbol.	8	0	3	t
8	identifier,	0	9	5	0
9	asymbol	0	0	1	t

It is assumed that for terminals (lines 3,4,5,7,9), the "son" field contains a pointer into some terminal symbol list. Thus, the first element of this list would contain "a", the second element "b", and the third a "+".

ANALYSIS RECORD

LINE #	POINT	SUP	SUB	PRED			j	COMMENT
1	1	0	0	0	1	2	1	initialization entry (find "expression")
2	2	1	0	0				
1			2					sub ← ν
					2	3		
								entry (find "identifier")
3	5	2	0	0				
2			3					sub ← ν
					3	4		
								entry (find "a")
							2	success ("asymbol" found) save ← 0
					2			
								success ("identifier" found)
					1			save ← 3
4	3	1	0	2				
1			4					sub ← ν
					4	5		
								entry (find "+")
							3	
								success ("+" found) save ← 4
					1			
5	4	1	0	4				
1			5					sub ← ν
					5	6		
								entry (find "b")
								failure ("b" not found)
1			4					sub_{sup} ← pred
					4	5		
								try again (find another "+")
							2	
								failure (uncover "+")
1			2					sub_{sup} ← pred
					2	4		
								try again (find another "identifier")
					3			λ ← sub
								try again (find another "a")
							1	
								failure (uncover "a")
3	6							point ← broth entry (find "identifier")
4	5	3	0	0				
3			4					sub ← ν
					4	5		
								entry (find "a")
							2	
								success ("a" found) save ← 0

TIME ↓

ANALYSIS RECORD CONT'D

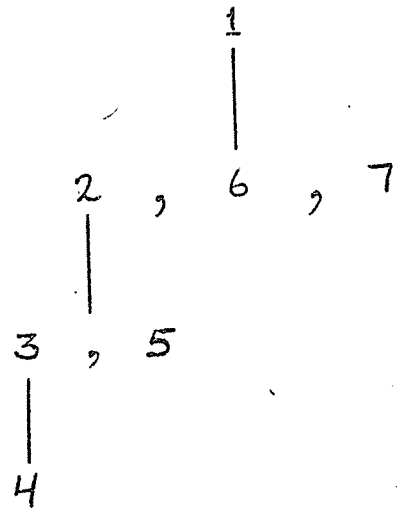
LINE #	POINT	SUP	SUB	PRED		j	COMMENT
					3		success ("identifier" found)
					2		save ⇐ 7
5	7	2	0	3			
2			5				sub ⇐ V
					5	6	entry (find "+" for "identifier")
						3	success ("+" found)
							save ⇐ 0
					2		success (identifier" found)
							save ⇐ 3
					1		
6	3	1	0	2			
1			6				sub ⇐ V
					6	7	entry (find "+")
						4	success ("+" found)
							save ⇐ 4
						1	
7	4	1	0	6			
1			7				sub ⇐ V
					7	8	entry (find "b")
						5	success ("b" found)
							save ⇐ 0
					1		sentence

TIME ↓

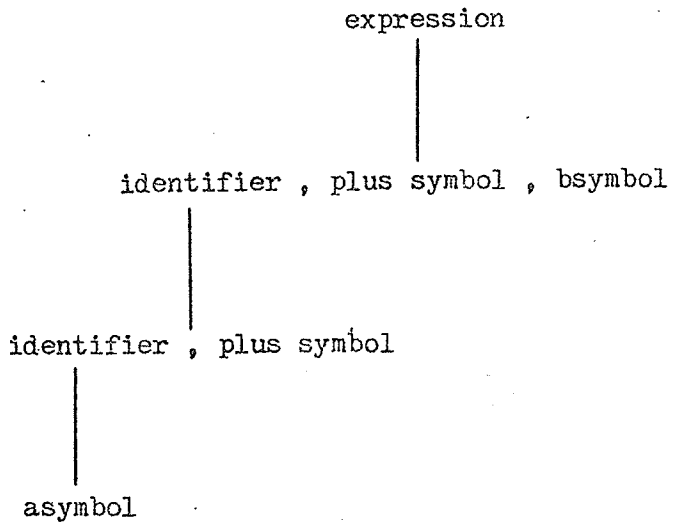
The final analysis record now has the following appearance:

LINE #	POINT	SUP	SUB	PRED
1	I	0	7	0
2	II	1	5	0
3	VI	3	4	0
4	V	3	0	0
5	VII	2	0	3
6	III	1	0	2
7	IV	1	0	6

Here, the Roman numerals are pointers into the syntax table whereas the Arabic numerals are links within the analysis record. If these links are placed on a tree similar to the definition tree, then one would obtain the following:



Replacing the Arabic numerals by the notions for which the Roman numerals of the corresponding line of the analysis record stand will then give a description of the analyzed sentence. This new tree would appear as follows:



Chapter 3

RECURSION

The principal difficulty with the parsing algorithm up to this point is its inability to handle a certain set of production rules which are known as recursive production rules.

In this chapter, recursion and some associated terms will be defined. This will be followed by a discussion of the difficulties these rules present to a top-down, left to right parsing algorithm. Following this, an algorithm that will enable the parsing algorithm to handle these rules will be presented.

3.1 BASIC DEFINITIONS

First, recall the difference between a notion and a node for the grammar G = (V, R, symbol, Z). A notion is a member of V whereas a node is a node of the

definition tree for G. Thus, A is a notion if it is in the set $\underline{V}$. If the production rules "B:A,C." and "D:A,E." are in $\underline{R}$, then A is a node within each production rule. Furthermore, the node A of the production rule "B:A,C." is quite distinct from the node A of the production rule "D:A,E.".

This distinction between notions and nodes should be kept in mind in the following discussions.

For the remainder of this chapter, a convention will again be adhered to. Capital letters, either subscripted or unsubscripted, will again represent notions. Underlined capital letters will represent sets. Additionally, any lower case letter, either subscripted or unsubscripted, from the set $\{u, v, w, x, y, z\}$ will represent some member of $\underline{V}^*$. They in fact represent notions separated by commas. Thus, "u" may for example represent the sequence "first notion, second notion, third notion".

Definition 3.1

A notion A in $\underline{V}$ is said to be a "recursive notion" if the relation $A \overset{*}{\Longrightarrow} A$, w. holds.

Definition 3.2

The ordered set of notions $\{A_0, A_1, A_2, \ldots, A_n\}$ where $n \geq 0$, is said to be a "cycle" if there exist rules in $\underline{R}$ which are of the form

$A_0 : A_1, w_1.$

$A_1 : A_2, w_2.$

$A_2 : A_3, w_3.$

.

.

.

$A_n : A_0, w_0.$

It should be noted that a certain ordering of notions exists in a cycle. Thus, in the above definition, $\{A_0, A_1, A_2, \ldots A_n\}$ is a cycle but $\{A_0, A_2, A_1, A_3, \ldots, A_n\}$ is not necessarily a cycle.

Note further that a cycle may possibly contain only one notion A as is the case when one of the production rules in R is of the form "A:A,w.". This type of production rule is usually called a left recursive production rule.

Lemma 3.1

If $\{A_0, A_1, \ldots, A_n\}$ is a cycle, then $\{A_i, A_{i+1}, \ldots, A_n, A_0, \ldots, A_{i-1}\}$ is a cycle.

Proof: By definition, there exist rules in R of the form

$A_i : A_{i+1}, w_{i+1}$.
$A_{i+1} : A_{i+2}, w_{i+2}$.
.
.
.
$A_n : A_0, w_0$.
$A_0 : A_1, w_1$.
.
.
.

$A_{i-1}: A_i , w_i .$

Hence, $\{A_i, A_{i+1}, \cdots , A_n, A_0, \cdots , A_{i-1}\}$ is a cycle by definition.

Definition 3.3

The cycles $\{A_1, A_2, \cdots , A_n\}$ and $A_i, A_{i+1}, \cdots, A_n, A_0, \cdots, A_{i-1}\}$ are said to be "equivalent". Thus, two cycles are "equivalent" if one can be obtained from the other by a cyclic permutation of its elements.

Definition 3.4

If "B:A,w." is in R then A is called an "entry notion" if A is in a cycle and the relation $A \overset{*}{\Longrightarrow} B,w.$ is false for all such B.

Definition 3.5

If "B:A,w." is in R, then this particular node A is called a "recursive node" if A is an entry notion and $A \overset{*}{\Longrightarrow} B,w.$ is false. We say that the recursive node A "invokes" the cycle in which A is an entry

notion.

Definition 3.6

If "$A_n : A_o , w_c .$" is in $\underline{R}$ then this particular occurrence of A is called a "cycle node" if A is an entry notion and $A_c \overset{*}{\Longrightarrow} A_n$, w. is true. We say that the cycle node A "reinvokes" the cycle to which A is an entry notion if the following conditions are met:

During the parse by the top-down, left-right parsing algorithm, the parse has proceeded along the lines

$$Z \overset{*}{\Longrightarrow} u_1, A_o, u_2. \overset{*}{\Longrightarrow} v_1, A_n, w, v_2.$$
$$\overset{*}{\Longrightarrow} v_1, A_o, w_c, w, v_2.$$

where $u_1 \overset{*}{\Longrightarrow} v_1$ and $u_2 \overset{*}{\Longrightarrow} v_2$.

The production rule that was applied in the direct production of v_1, A_o, w_o, w, v_2 from v_1, A_n, w, v_2 is obviously "$A_n : A_o, w_o.$" and A_o will therefore reinvoke the cycle to which A_o is an entry notion.

It should be noted that it is quite possible to apply a production rule which contains a cycle node on the RHS without that cycle node reinvoking a cycle. Example 3.1 illustrates this point.

Theorem 3.1

If an input string u satisfies the nth partition of a recursive node A, then the ith partition of A will be satisfied in u for all $0 \leq i \leq n$.

Proof: By n successive applications of sequences of production rules to A, we have $A \overset{*}{\Rightarrow} A, w_1 \cdot \overset{*}{\Rightarrow} A, w_2, w_1 \cdot \overset{*}{\Rightarrow} \ldots \overset{*}{\Rightarrow} A, w_n, w_{n-1}, \ldots, w_1 \cdot \overset{*}{\Rightarrow} w_{n+1}, w_n, \ldots, w_1 \cdot \overset{*}{\Rightarrow} u$. by statement of the theorem. But, $A \overset{*}{\Rightarrow} A, w_2 \cdot \overset{*}{\Rightarrow} A, w_3, w_2 \cdot \overset{*}{\Rightarrow} \ldots \overset{*}{\Rightarrow} A, w_n, w_{n-1}, \ldots, w_2 \cdot \overset{*}{\Rightarrow} w_{n+1}, w_n, \ldots, w_2 \cdot$ is therefore also satisfied in u and this is clearly in the nth-1 partition of A. This proves the theorem by induction.

Example 3.1

Suppose a parse had proceeded along the lines

$Z \Longrightarrow B_1, A_i, B_2. \overset{*}{\Longrightarrow} B_1, A_n, w, B_2.$

$\Longrightarrow B_1, A_0, w_0, w, B_2.$ where $i \geq 1$. Suppose further that the grammar contains a cycle of the form $\{A_0, A_1, \ldots, A_n\}$ and that A_0 is an entry notion to the cycle. Then in the above parse, the last production rule applied was again $A_n : A_0, w_0.$ However, this A_0 does not reinvoke the cycle for which A_0 is an entry notion since the parse has not proceeded along the required lines; ie. the cycle has not been previously invoked via the entry notion A_0.

To state matters in simpler language, a cycle node A_0 only reinvokes the cycle to which A_0 is an entry notion if the parser has previously entered the cycle by making A_0 its goal. In trying to satisfy this goal A_0, it may then apply a sequence of productions which will lead the parser to consider A_n as a goal without any "symbols" being recognized in the meantime. If the rule $A_n : A_0, w_0.$ is then applied, the parser will again consider A_0 as its goal and it is at this

stage that the cycle is said to be reinvoked.

It is now easy to see that should the parser proceed along these lines, it will make A_o a subgoal which will make A_o a subgoal which will ... ad infinitum. Thus, the parser will be caught in a loop from which it can not extricate itself unless specific provisions are made to stop this looping.

This infinite looping is precisely the problem that recursive production rules present to top-down left-right parsing algorithms.

Definition 3.7

If A_o is a recursive node in the rule B : w_1, A_o, w_2. and $A_o \stackrel{*}{\Longrightarrow} u.$ where "u" is a string of symbols, then we say that "u" is in the n^{th} partition of this particular recursive node A_o if

$A_o \stackrel{*}{\Longrightarrow} A_o, v_1. \stackrel{*}{\Longrightarrow} A_o, v_2, v_1.$

$\stackrel{*}{\Longrightarrow} A_o, v_3, v_2, v_1. \stackrel{*}{\Longrightarrow} \ldots$

$\stackrel{*}{\Longrightarrow} A_o, v_n, v_{n-1}, \ldots, v_1. \stackrel{*}{\Longrightarrow} u.$ is true, and

A_0 , v_n, v_{n-1} , ... v_1. $\stackrel{*}{\Longrightarrow}$ A_0, v_{n+1}, v_n, ..., v_1.

$\stackrel{*}{\Longrightarrow}$u. is false.

In simpler terms, "u" is in the n^{th} partition of the recursive node A_0 if, in determining an input string to satisfy A_0, A_0 had to be made the subgoal of the parsing algorithm exactly n times before any recognition of characters in the input string took place.

Note that "u" may be in partition zero of A_0. This occurs when R for example contains the production rules

A: B, A_0, C.

A_0: A_0, D; E.

and, in recognizing an A_0, the rule "A_0 :E." was applied.

Following will be some examples that will consolidate the definitions presented so far.

Example 3.2

Let R be the set

"prog: id." , "id: id, letter; letter." ,
"letter: asymbol."

Then,

the set {id} is a cycle.

the notion "id" is a recursive notion.

the notion "id" is an entry notion.

in "prog: id.", "id" is a recursive node which invokes the cycle {id} .

in "id: id, letter; letter.", the node "id" on the RHS is a cycle node which may reinvoke the cycle {id} depending on the history of a particular parse. In the above example, this cycle node will always reinvoke the cycle {id} irregardless of the history of the parse.

Example 3.3

Let R be the set

"prog:id." , "id: idl." , "idl: id, letter; letter." , "letter: asymbol."

Then,

the set {id, idl} is a cycle.

the set {idl, id} is a cycle which is equivalent to the cycle {id, idl} .

the notion "id" is a recursive notion.

the notion "idl" is a recursive notion.

the notion "id" is an entry notion but "idl" is not.

in "prog: id.", "id" is a recursive node which invokes the cycle {id, idl} . Note that in "id: idl.", "idl" is not a recursive node.

in "id1: id, letter; letter.", the node "id" is a cycle node which may reinvoke the cycle {id, id1} depending on the history of a particular parse. In this example, this cycle node will always reinvoke the cycle {id, id1} irregardless of the history of the parse.

Example 3.4

let R be the set of rules

{prog: a0.
a0: a1; wsymbol1, b1.
a1: a2, xsymbol.
a2: a0, ysymbol.
b1: a2, zsymbol.}

Then,

there are 3 cycles all equivalent to {a0, a1, a2} .

the notions "a0", "a1", and "a2" are recursive notions.

the notions "a0" and "a2" are entry notions but "a1" is not.

in "prog: a0.", a0 is a recursive node which invokes the cycle {a0, a1, a2} .

in "b1: a2, zsymbol.", a2 is a recursive node which invokes the cycle {a2, a0, a1} .

in "a2: a0, ysymbol.", a0 is a cycle node which may reinvoke the cycle {a0, a1, a2} depending on the history of a particular parse.

in a2: ao, ysymbol.", a2 is a cycle node which may reinvoke the cycle {a2, a0, a1} depending on the history of a particular parse.

Suppose a particular parse has proceeded along the line

prog $\Longrightarrow$ a0. $\Longrightarrow$ a1. $\Longrightarrow$ a2, xsymbol.

In this case, the last direct production was effected using the rule "a1:a2,xsymbol." where

"a2" is a cycle node. However, in this case, the cycle node a2 does not reinvoke the cycle {a2, a0, a1} because it was not invoked previously by application of the rule "b1:a2,zsymbol." We also say that the cycle node a2 is "not active".

Suppose the parse had proceeded

Prog⟹a0.⟹wsymbol, b1.
⟹wsymbol, a2, zsymbol.
⟹wsymbol, a0, ysymbol, zsymbol.
⟹wsymbol, a1, ysymbol, zsymbol.
⟹wsymbol, a2, xsymbol, ysymbol, zsymbol.

Then in "wsymbol, b1⟹wsymbol, a2, zsymbol.", the production rule "b1:a2, zsymbol.", would have been applied and we would say that the recursive node "a2" would have invoked the cycle {a2, a0, a1} . We also say in this case that the cycle node a2 is "active".

In

wsymbol, a1, ysymbol,zsymbol.

$\Longrightarrow$wsymbol, a2, xsymbol, ysymbol, zsymbol,

the production rule "a1:a2,xsymbol." would have been applied and we would then say that the cycle node "a2" reinvoked the cycle {a2, a0, a1}.

Example 3.5

Let R be the set of rules

{a: b, semicolon symbol.

b: b_1, asymbol; c , asymbol; asymbol.

b_1: b, bsymbol.

c: b, csymbol.

Then,

there are two cycles which are not equivalent. These are $\{b, b_i\}$ and $\{b, c\}$.

the notions "b", "b_1", and "c" are recursive notions.

the notion "b" is an entry notion.

in"a:b,semicolonsymbol.", "b" is a recursive node which invokes one of the cycles $\{b, b_i\}$ or $\{b, c\}$. In fact, b may invoke both cycles at the same time as will be shown below.

in "b_i :b,bsymbol." "b" is a cycle node which may reinvoke the cycle

$\{b, b_i\}$ or the cycle $\{b, c\}$

in "c:b,csymbol.", "b" is a cycle node which may reinvoke the cycle $\{b, c\}$ or the cycle $\{b, b_1\}$.

Of particular importance in this case is the

fact that only one entry notion exists for both cycles and that both cycles are entered through this entry notion. In fact, in "a:b,semicolon symbol.", the recursive node b may be considered as invoking both cycles $\{b, b_1\}$ and $\{b, c\}$ at the same time.

A further question now arises in what is meant when we say that a terminal string "u " is in the nth partition of the recursive node b. In fact the definition for "nth partition" reveals that, if the parser traverses the cycle $\{b, b_1\}$ "i" times and the cycle $\{b, c\}$ "j" times in the recognition of b, then "u" is in the nth partition of the recursive node b where n=i+j.

The reason for this is of course that, in having the recursive node b as its goal, the parser may make the sequence of notions "b_1 , b" and " c , b" its subgoals in any random fashion.

It now becomes useful to classify cycles according to 3 different types. These are the following:

Type 1:

If $\{A_0, A_1, \ldots, A_n\}$ is a cycle, then A_0 is the only entry notion to it. Also, $\{A_0, A_1, \ldots, A_n\}$ is the only cycle with entry notion A_0.

Type 2:

If $\{A_0, A_1, \ldots, A_n\}$ is a cycle, then there are at least two entry notions A_i, A_j. Also $\{A_0, A_1, \ldots, A_n\}$ is the only cycle with entry notions A_i, A_j.

Type 3:

If $\{A_0, A_1, \ldots, A_n\}$ is a cycle then A_0 is the only entry notion to the cycle. However, there are other distinct cycles with A_0 also being their only entry notion.

All other more complicated cycles not of these types may be constructed from these first 3 types.

This completes the definitions required for the discussion of recursion. The problem that recursive production rules present to a top-down left to right parsing algorithm have been touched upon previously and will now be stated explicitly.

If A_0 is an entry notion in the grammar G = ($\underline{V}$,$\underline{R}$, symbol, Z) then a conventional top-down left to right parsing algorithm may be caught in a never ending loop if it should have A_0 as its goal. This problem will occur if the subgoals of the parsing algorithm are successively the recursive notions of a cycle in which A_0 is an entry notion. Thus, if $\{A_c, A_1, \ldots, A_n\}$ is such a cycle, the parsing algorithm may have successive subgoals of $A_1, A_2, \ldots, A_n$ and application of the rule A_n: A_c, w_c. will then cause the parsing algorithm to again select A_c as a goal. Thus, the parser will be "trapped" in this cycle of subgoals.

3.2 AN ALGORITHM FOR HANDLING RECURSION

A method for stopping this looping will now be

presented. Recall the format of the syntax list which contains the syntax definitions for the grammar. Each node of the R.H.S. of a production rule was entered as a line in this syntax list and each such node had a code field available to it.

We will now adopt the convention of placing a special "recursive code" into the code field of a recursive node of the grammar, and a "cycle code" into the code field of a cycle node of the grammar. As was mentioned previously, these code fields may be pointers which point to routines that carry out special actions for the distinct codes.

Suppose we have the grammar of example 1 where R is the following set of productions:

{prog: id.

id: id, letter; letter.

letter: asymbol.}

This grammar obviously contains only the type 1 cycle {id} . The syntax table for this grammar would now appear as follows:

SYNTAX TABLE

	broth	comp	son	code
1. prog	0	0	2	0
2. id.	0	0	3	r
3. id,	5	4	3	c
4. letter;	5	0	6	0
5. letter.	0	0	6	0
6. asymbol.	0	0	1	t

The codes "r" and "c" stand for "recursive" and "cycle" respectively. It should be noted that the entry notion "id" is defined in line 3. Furthermore, the sons of both the lines for the recursive node and the cycle node indicate line 3. This state of affairs is true for any general cycle; the son of the line for the recursive node will always be the same as the son of the line for the cycle node. The line that is indicated by these sons will always be the line where

the entry notion to the cycle is defined.

This extremely useful property will now be exploited. During a parse, if a recursive node is made a goal of the parsing algorithm, a seven element data area is created. This area will be tagged with the son of the recursive node, in this example this son would be 3. The cycle node can then refer to this stack also through its son, namely 3. This effectively allows some means of counting how many times a cycle has been traversed. It is only necessary to communicate with the stack element created for the recursive node whenever the corresponding cycle node is made a goal. It is also possible at this stage to carry out certain special actions, such as for instance disallowing reinvocation of the cycle to prevent infinite looping.

3.2.1 Recursion Handling Data Areas

There are two data areas required for the recursion handling.

1. A list consisting of seven elements per line. Each line has the following format:

rsup	rsub	rpred	part	succ	level	rj

"rsup","rsub", and "rpred" are pointers internal to this recursion list. They are similar to the superior, subordinate, and predecessor of the analysis record in that they are responsible for maintaining the structure of the list.

"rj" is an area which contains a stored value of the input string pointer "j",

"part" is a location which stores a partition value, and

"succ" and "level" are two switches which hold the value "true" or "false".

2. A list known as the "recursion pointer list", of two pointers of the following format:

"stp" is a syntax table pointer. It corresponds exactly to a line of the syntax table where an entry notion is defined.

"rlp" is a pointer into the recursion list which indicates the last line in the recursion list with which the particular entry notion indicated by "stp" was associated.

Recall that, in the syntax table, the son of a recursive node and the son of the corresponding cycle node were identical to the line number where the definition for an entry notion began. It is exactly this number which is recorded in "stp" above. The corresponding "rlp" points to the seven element entry in the recursion list which was created when that

particular entry node was made a goal of the parser.

Since any particular cycle may be invoked many times during a particular parse, it may be necessary to have more than one line entered in the recursion list. Thus, since "rlp" will always point only at the last such line, the fields "rsup", "rsub", and "rpred" are used to chain all the lines for a particular recursive node together in much the same way in which the lines of the analysis record are chained together by the "superior", "subordinate", and "predecessor" fields of the analysis record.

It is necessary at this stage to explain the use of "part", "succ", and "level" of the recursion list. Recall the definition of the "nth partition of a recursive node A". By this is meant the number of times a cycle, to which A is an entry notion, is reinvoked during a parse. Being an important quantity, this number is kept in the "part" or "partition" field of the recursion list.

The "partition" field is initially set to zero, the "success" field to "false", and the "level" field to true. The meaning of the contents of these fields

is as follows:

1. "partition" initially indicates the partition of the recursive node that is to be examined. Thus, if the field contains the value "n" when the corresponding recursive node is made a goal (either directly or with a try again command), then this indicates that a report of success is to be accepted only from the nth partition of the recursive node. The partition field also doubles up as a counter when the cycle is traversed. Every time the cycle is reinvoked the partition field is decremented by one. Therefore, if this field reaches zero, it indicates that the nth partition of the recursive node is being scanned. On the other hand, if success or failure is reported for a cycle node, the partition field is incremented by one in order to retain the integrity of this field.

2. "success" is initially set to "false". If success is reported for the recursive node from the nth partition, "success" is set to "true" indicating that success may possibly be found for the recursive node in the nth + 1 position.

3. "level" is initially set to "false" if the partition field is greater than zero. The level field is set to "true" when success is reported for a cycle node whose partition contains zero. It thus serves as an indicator when success is reported for a recursive node whether or not the report of success comes from the nth partition.

This gives a broad description of the use of the data areas of the recursion list. A better understanding of these areas and their function can be obtained from the following discussions on the modification of Floyd's algorithm that will provide this algorithm with recursion handling capabilities.

To this end, Floyd's algorithm must be modified in four locations: after "entry", after "success", after "failure", and after "try again" where the quantities indicated represent labels of the flowchart given on page 32.b. It is assumed that these modifications will be in the form of insertions of code or flowchart sections immediately after the respective labels, with the exception of the section of the flowchart dealing with success. The reason is that success is not reported "for" a node but "to" a "superior" node. Any next components of the recognized node must thus be recognized before success can be reported "for" the "superior" node. A modified flowchart is given on page 79b . Discussion of each modification will be subdivided under two subheadings:

a) action taken when the goal is a recursive node

b) action taken when the goal is a cycle node.

3.2.2 Modifications to "entry"

a) The goal is a recursive node.

The action is simply to generate a new line in the recursion list and associate it with this recursive node. The data elements of the line are initialized as previously indicated.

b) The goal is a cycle node.

i) The "partition" corresponding to the cycle node is decremented by one.

ii) The "partition" field is checked. If it is less than zero, then the cycle should not have been reinvoked and failure is reported. If the "partition" field is greater than or equal to zero, then execution continues normally.

3.2.3 Modifications to "success"

a) The goal for which success was reported is a recursive node.

i) The "level" field of the recursion list line associated with the recursive node is checked. If it is "true", then success has been reported for the nth partition and the report of success is accepted. In this case, the "success" field of the recursion list line is set to "true". The recursion list line corresponding to the recursive node is disassociated from the recursive node via the "rsup", "rsub", and "rpred" fields of the recursion list line. Details of this operation are presented later.

ii) If the "level" field is "false", then the nth partition of the recursive node had not been reached. The report of success will not be accepted and a "try again" command is issued. It is this device that ensures that

the only reports of success that will be accepted for a recursive node will be reports of success in the nth partition.

b) The goal for which success was reported is a cycle node.

i) The "partition" field corresponding to the cycle node is checked. If it is zero, then the report of success is in the nth partition of the corresponding recursive node. To communicate this fact to the recursive node to which success will (potentially) be reported at a later stage, the "level" field of the recursion list line is set to "true". The "partition" field is incremented by one and the parser continues normally.

ii) If the "partition" field is not zero, the "level" field is checked. If the "level" field is "true", then the report of success is accepted and the partition field is incremented

by one before continuing normally. If the "level" field is false, then obviously the nth partition of the recursive node has not been reached. The report of success is thus rejected and a "try again" command is issued.

3.2.4 Modifications to "failure"

a) The goal for which failure is reported is a recursive node.

A report of failure at this stage indicates that the entire nth partition of the recursive node has been scanned without success. However, it is possible that success had previously been reported in the nth partition of the recursive node and that the current report of failure is the result of a "try again" command to the recursive node. In that case, there is a possibility of satisfying the recursive node in the nth + 1 partition. Thus, one of the

following two groups of actions are carried out:

i) If the "success" field is "true", the "partition" field is incremented by one, the "success" field and the "level" field are set to "false", and the parser continues normally as if the recursive node had been made a goal for the first time. These actions effectively initiate the parser for an attempt to satisfy the recursive node in its nth + 1 partition.

ii) If the "success" field is "false", then this indicates that the entire nth partition of the recursive node had been scanned without a report of success. There is therefore no possibility that success for the recursive node will be found in the nth + 1 or a higher partition. The report of failure for the recursive node will thus be accepted. Additionally, the corresponding recursion list line is no longer required for the recursive node and it is thus returned to a "free

recursion line list".

b) The goal for which failure is reported is a cycle node.

The report of failure is accepted and the partition field is incremented by one.

3.2.5 Modifications to "try again"

a) The goal which is told to try again is a recursive node.

The recursion list line which was previously associated with the recursive node is reestablished. For this purpose, the "rsup", "rsub", and "rpred" fields of the recursion list line are provided. Again, the details of their use will be explained at a later stage (see 4.2). Another try again command is then issued.

b) The goal which is told to try again is a cycle node.

The only actions here are that the partition field is decremented by one and that the "level" field is set to "false". The reason for setting the "level" field "false" at this stage is as follows:

When a recursive node is told to "try again", the "level" field is still "true" from the previous report of success for the recursive node. If the "partition" field happens to be zero, then the "level" field should contain "true". If the field happens to be greater than zero, however, the level field should be set to "false". This will automatically be done by the "try again" command to the first cycle node which is told to "try again".

A second instance where the "level" field must

be reset to "false" on a "try again" command to a cycle node occurs when the nth partition of a recursive node has been reached with some partial success being reported in this partition. The "level" field will thus be set to "true". However, it may well happen that this partial report of success is later found to be unsuitable because some component could not be recognized. "Try again" commands will thus be issued. These commands should then reset the "level" field to "false" when a cycle node is told to "try again". Otherwise, there is a distinct possibility that a report of success in the nth - 1 partition would be accepted since the "level" field incorrectly remained "true". This could quite possibly cause the parser to go into an infinite loop because a "try again" command for the nth partition will return a report of success for the nth - 1 partition which will subsequently be found unsuitable and result in a "try again" command for the nth partition ... etc... ad infinitum.

It is well to summarize the effects of these modifications in order to bring their purpose back into focus. They basically carry out three distinct actions.

1. A report of success for a recursive node at any one time is accepted only for the nth partition of that recursive node.

2. The entire nth partition of a recursive node is tried before the nth + 1 partition is tried.

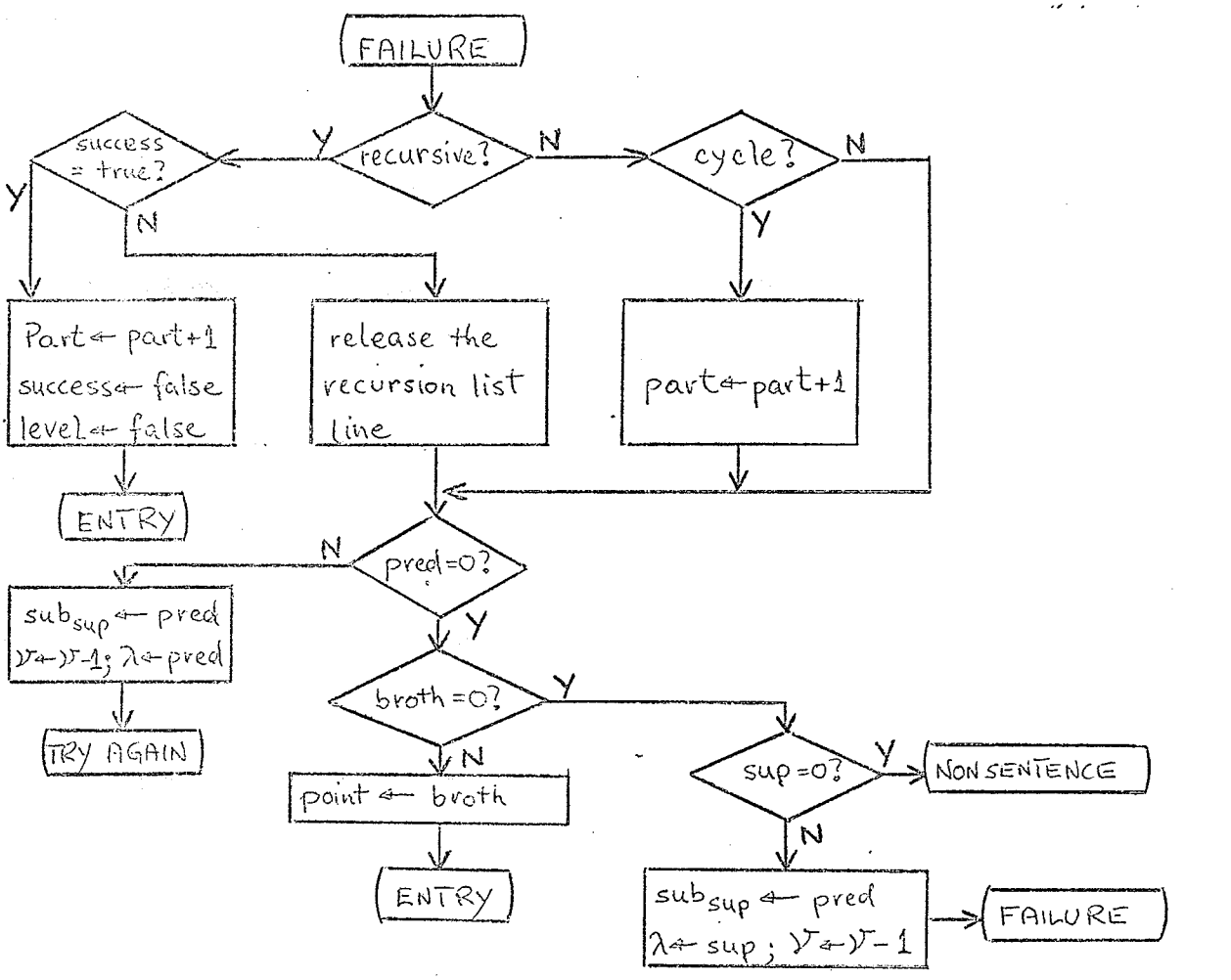
3. The nth + 1 partition of a recursive node is tried only if success had been previously reported in the nth partition and a subsequent try again command to the recursive node failed to produce success in the remainder of the nth partition of the recursive node.

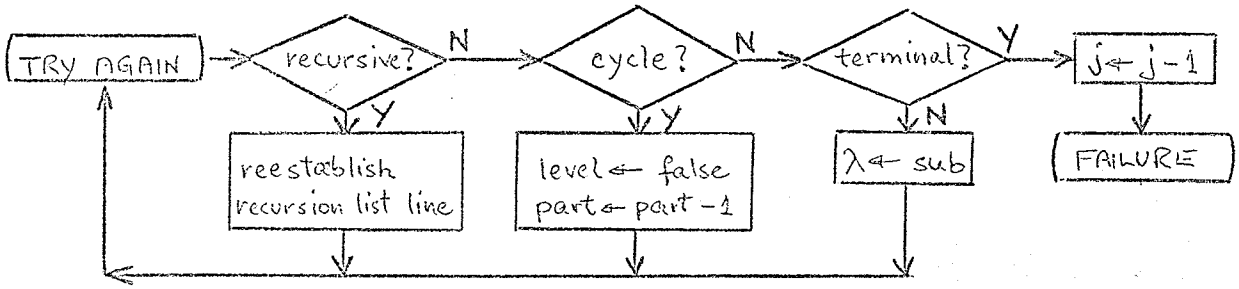
It is obvious that the method of handling recursion presented so far will easily handle all type 1 cycles of a grammar as long as these type 1 cycles

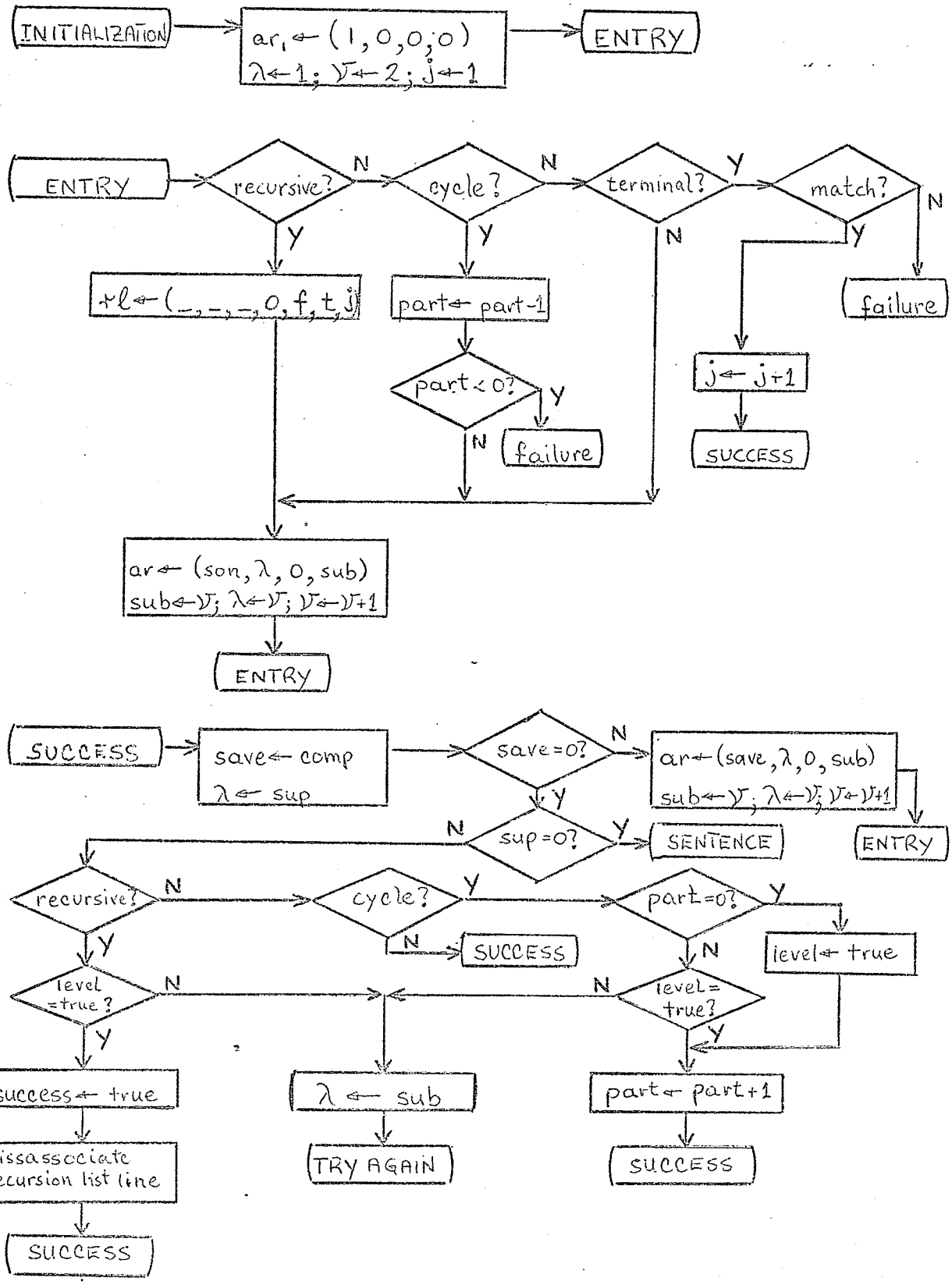
are not interconnected in some way.

There are however some difficulties involved with type 2 cycles and cycles that interconnect in some way. These will be discussed in the following chapter.

To complete this chapter, a flowchart of Floyd's algorithm with the modifications mentioned in this chapter follows.


FAILURE
recursive?
Y
N
success = true?
Y
N
cycle?
N
Y
Part ← part+1
success ← false
level ← false
release the recursion list line
part ← part+1
ENTRY
pred=0?
N
Y
sub_sup ← pred
ν ← ν-1; λ ← pred
TRY AGAIN
broth=0?
Y
N
point ← broth
ENTRY
sup=0?
Y
NON SENTENCE
N
sub_sup ← pred
λ ← sup ; ν ← ν-1
FAILURE


TRY AGAIN
recursive?
N
Y
cycle?
N
Y
terminal?
Y
N
j ← j-1
reestablish recursion list line
level ← false
part ← part-1
λ ← sub
FAILURE


INITIALIZATION
ar1 ← (1,0,0,0)
λ ← 1; V ← 2; j ← 1
ENTRY
ENTRY
recursive?
N
cycle?
N
terminal?
Y
match?
N
Y
Y
N
Y
rl ← (_,_,_,0,f,t,j)
part ← part-1
failure
part < 0?
Y
N
failure
j ← j+1
SUCCESS
ar ← (son, λ, 0, sub)
sub ← V; λ ← V; V ← V+1
ENTRY
SUCCESS
save ← comp
λ ← sup
save = 0?
N
Y
ar ← (save, λ, 0, sub)
sub ← V; λ ← V; V ← V+1
ENTRY
N
sup = 0?
Y
SENTENCE
recursive?
N
Y
cycle?
Y
N
SUCCESS
part = 0?
Y
N
level ← true
level = true?
N
Y
N
level = true?
Y
success ← true
dissassociate recursion list line
SUCCESS
λ ← sub
TRY AGAIN
part ← part+1
SUCCESS

Chapter 4

A GENERAL RECURSION HANDLING ALGORITHM

We have seen in the previous chapter how simple type 1 cycles may be handled. In this chapter, the handling of type 2 and type 3 cycles will be discussed. This will be followed by a description of some routines that are necessary to do proper housekeeping of the recursion list lines. Following this, a new flowchart including these modifications will be presented and this will then in effect be the complete, general parsing algorithm for context-free grammars.

4.1 HANDLING TYPE 2 AND TYPE 3 CYCLES

The algorithm presented in Chapter 3 for handling type 1 cycles is also sufficient for type 3 cycles and needs no modification to handle these

cycles. Type 3 cycles will therefore not be further discussed.

Type 2 cycles however do require a slight modification of the recursion handling algorithm. Recall that type 2 cycles have more than one entry notion and therefore have more than one cycle notion of which only one is active for any invocation of the cycle. For example, suppose $\{A_0, A_1, \ldots, A_i^*, \ldots A_n\}$ is a cycle with entry notions A_0 and A_i^*. Thus, there exist production rules

A_n: A_0, w_0. where A_0 is a cycle node and

A_{i-1}^*: A_i^*, w_i^*. where A_i^* is a cycle node.

Suppose further that the cycle was initially invoked via entry notion A_0. Thus, A_i^* is not an active cycle node, i.e. it does not reinvoke a cycle since no such cycle was initially invoked via entry notion A_i^*.

However, in attempting to satisfy the recursive node A_0, the production rule A_{i-1}^*: A_i^*, w_i^*. may well be

applied at some stage and the parser is now faced with the problem of recognizing that A_i is not an active cycle node. If this is not recognized, then it is possible that some data areas of the recursion list line associated with some previous invocation of the cycle via the entry notion A_i will be incorrectly altered.

Fortunately, it is a relatively simple matter to determine whether a given cycle node is active or not. When a cycle node A_i is made a goal, it is first of all determined if there is some recursion list line associated with the entry notion A_i. If not, then we can immediately say that the cycle node A_i is not active. If there is a recursion list line for the entry notion A_i, the stored input string pointer value "rj", stored when a recursive node A_i was previously made a goal, is compared with the current value of the input string pointer "j". If these two are equal, then obviously the parser has proceeded along a parse $A_i \overset{*}{\Longrightarrow} A_i$, w. and thus the cycle node is active since the cycle is reinvoked. If the two pointer values are not the same, then the parser has obviously proceeded along a parse $A_i \overset{*}{\Longrightarrow} w_1$, A_i, w_2. and the A_i of the r.h.s. obviously does not reinvoke the cycle, i.e. it

is not active.

It is now necessary to record whether or not a given cycle node is active. This will be done by extending the analysis record to include a fifth entry for each line, called the "active" field. This will be set to "true" if an active cycle node is made a goal and to "false" if an inactive cycle node is made a goal. All goals other than cycle nodes ignore the "active" field. To demonstrate the need for this field, consider the following situation:

> A cycle node is made a goal and subsequently success is reported for it. If the cycle node was originally active when made a goal, then its "partition" field must be incremented. If it was not active originally, then no special action must be carried out and it must be handled just as an ordinary node. Therefore it must be known whether or not the cycle node is active. This can no longer be determined by comparing an "rj" value with a current input string pointer value because the input string pointer will have a different value than it had

when the cycle node was originally made a goal. The fact that success has been reported for the cycle node indicates that some recognition took place in the input string and the input string pointer will thus have been advanced.

The above example demonstrates the necessity for storing an activity indicator for a cycle node when it is first made a goal.

There are therefore a few minor changes to the flowchart of page 79b in order to allow the algorithm to handle type 2 grammars. These are

1. In the entry portion of the flowchart, a test to determine whether or not a cycle node is active must be inserted immediately after the decision that the goal is a cycle node. If the cycle node is active, then the "active" field of the analysis record must be set to "true"; if the cycle was not active, this field must be set to "false".

2. The decisions of whether or not a goal is or was a cycle node should be changed to read "active cycle?" instead of "cycle?" in the "success", "failure", and "try again" portion of the flowchart.

4.2 ADMINISTRATION OF THE RECURSION LIST

The remaining portion of the recursion handling algorithm which has as yet not been explained is the administration of the recursion list lines. A short description of the organization of the recursion list and the recursion pointer list is required.

As was mentioned previously, the recursion pointer list is a list of all recursive notions of a grammar. For each recursive notion, there is a pointer field "rlp" which points to the recursion list line currently associated with the recursive notion.

Initially, there will be one recursion list line associated with each recursive notion and "rlp" will contain a link to this line. The "rsup" field of all these lines is initially set to zero to denote that this particular line is merely a dummy entry required for administration purposes.

The unused lines of the recursion list are chained together via the "rsub" field and a global variable contains a pointer to the first of these chained lines. Obtaining unused lines and returning used lines from and to this "empty list" is therefore easily carried out.

The "rlp" field of the "recursion pointer list" will always contain a pointer to the recursion list line associated with a currently active cycle. Since there may be other entries in the recursion list for a particular recursive notion, these entries are chained together via the "rsup", "rsub", and "rpred" field of the recursion list lines.

Administration of the recursion list and

recursion pointer list may now be divided into four logical groupings:

1. Pushdown.

   This operation is carried out whenever a new recursive node is made a goal of the parser. "Pushdown" consists of obtaining a new recursion list line from the "empty list", proper initialization of this line, and the carrying out of administrative functions.

2. Popup.

   This operation is carried out when failure is reported for a recursive node throughout an entire partition. The recursion list line is returned to the "empty list" and administrative functions are carried out.

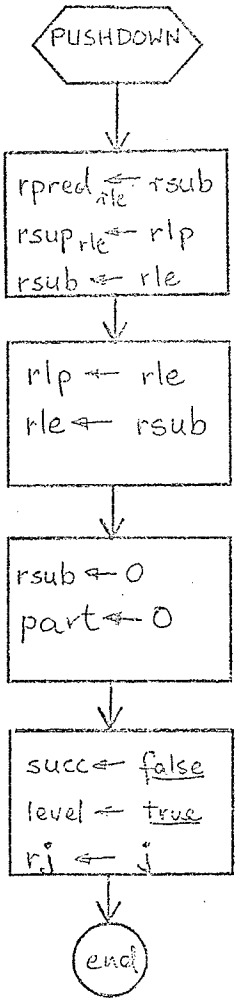
3. Setdown.

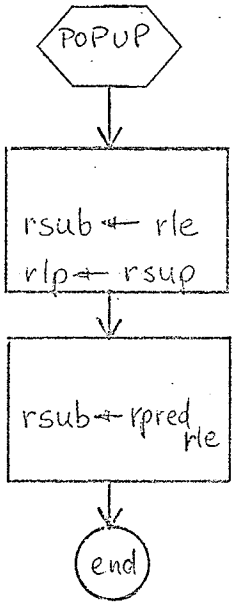
This operation is carried out when a recursive node is told to "try again". Setdown will find the recursion list line previously associated with this recursive node and place a pointer to this line in the proper "rlp" field of the recursion pointer list.

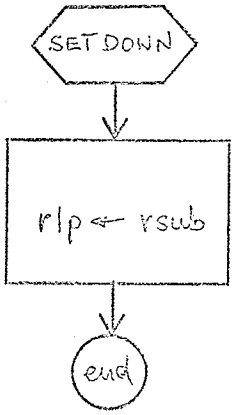
4. Setup.

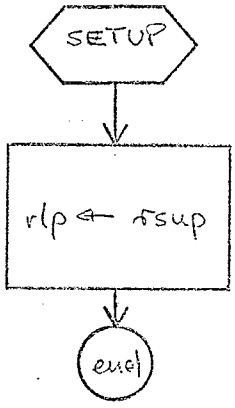
This operation is carried out when success is reported for a recursive node. It causes the recursion list line to be saved so that it may be found again by a subsequent "setdown" operation.

The detailed operation of these routines is given in the flowchart that follows. The terminology is as follows:


PUSHDOWN
rpred rle ← rsub
rsup rle ← rlp
rsub ← rle
rlp ← rle
rle ← rsub
rsub ← 0
part ← 0
succ ← false
level ← true
ri ← i
end


POPUP
rsub ← rle
rlp ← rsup
rsub ← rpred rle
end


SETDOWN
rlp ← rsub
end


SETUP
rlp ← rsup
end

1. "rlp" stands for the contents of the "rlp" field of the entry of the recursion pointer list associated with the recursive notion currently under consideration by the parser.

2. "rle" contains a pointer indicating the next empty line of the recursion list.

3. Quantities "rsup", "rsub", "rpred", "part", "succ", "level", and "rj" of the recursion list are always implicitly subscripted by "rlp" unless explicitly subscripted otherwise.

4. "j" is the input string pointer.

The similarity between the administration of the recursion list and the administration of the analysis record is quite evident. It is useful now to go through an example demonstrating how these operations work in the recursion list.

Consider a grammar with a cycle whose entry notion is A. Suppose further that the following rule exists in the grammar:

$$Z: w_1, A, w_2. \text{ where } w_2 \overset{*}{\Longrightarrow} w_3, A, w_4. .$$

Then, when Z is made a goal of the parser and after w_1 has been recognized, the recursive node A of this rule will be made a goal. When this happens, a "pushdown" is executed and the recursion list line for "A" is recorded as a subordinate in the "rsub" field of the previous recursion list line for "A". After an "A" has been recognized, a "setup" operation is executed which will cause "rlp" to be reset to this previous recursion list line.

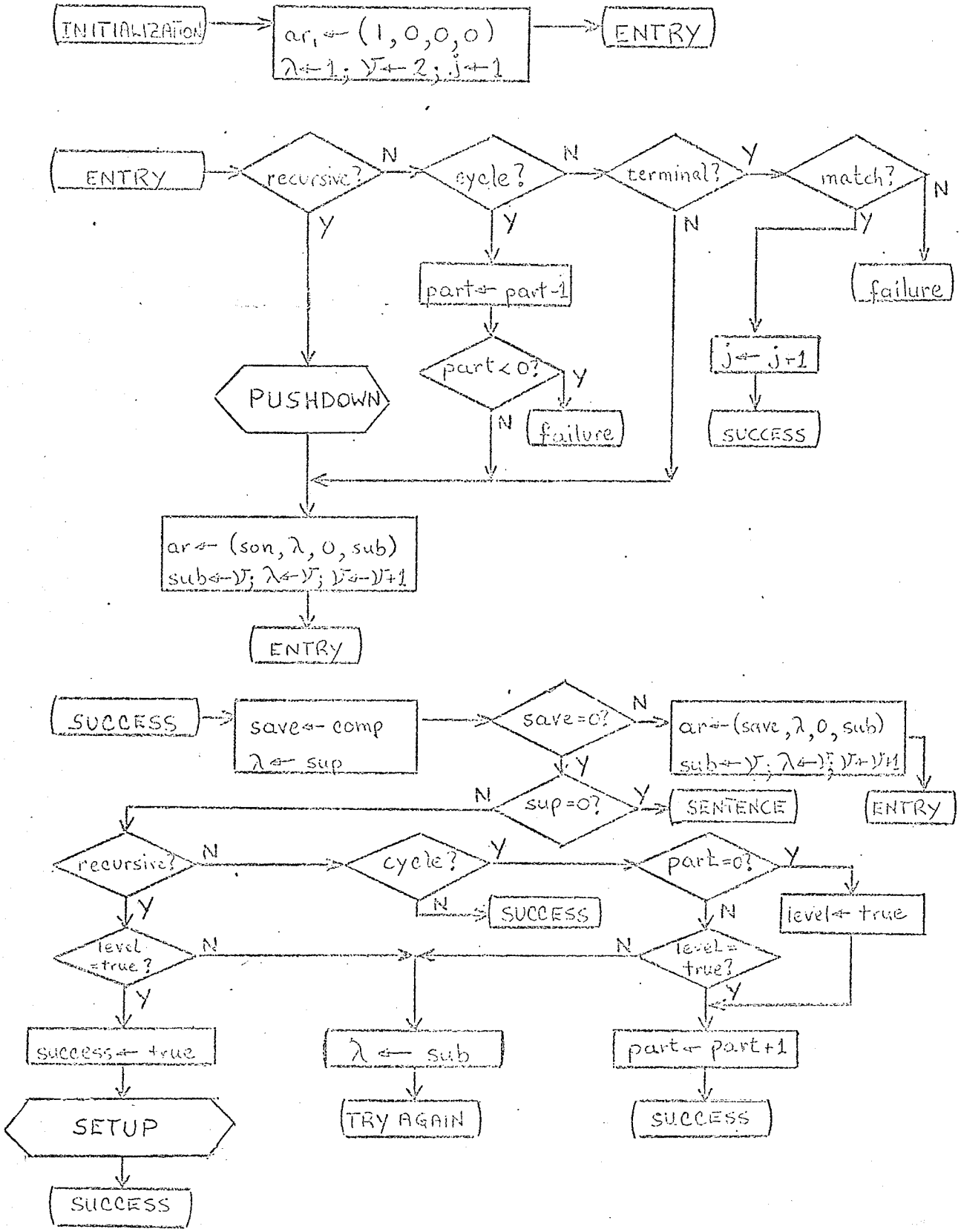
During the subsequent recognition of w_2, the node "A" in "w_3, A, w_4." will again reinvoke a cycle and a new "pushdown" operation will occur. The "rsub" field of the initial recursion list line will be set to point to this new recursion list line. The "rpred" field of this new recursion list line will thus be set

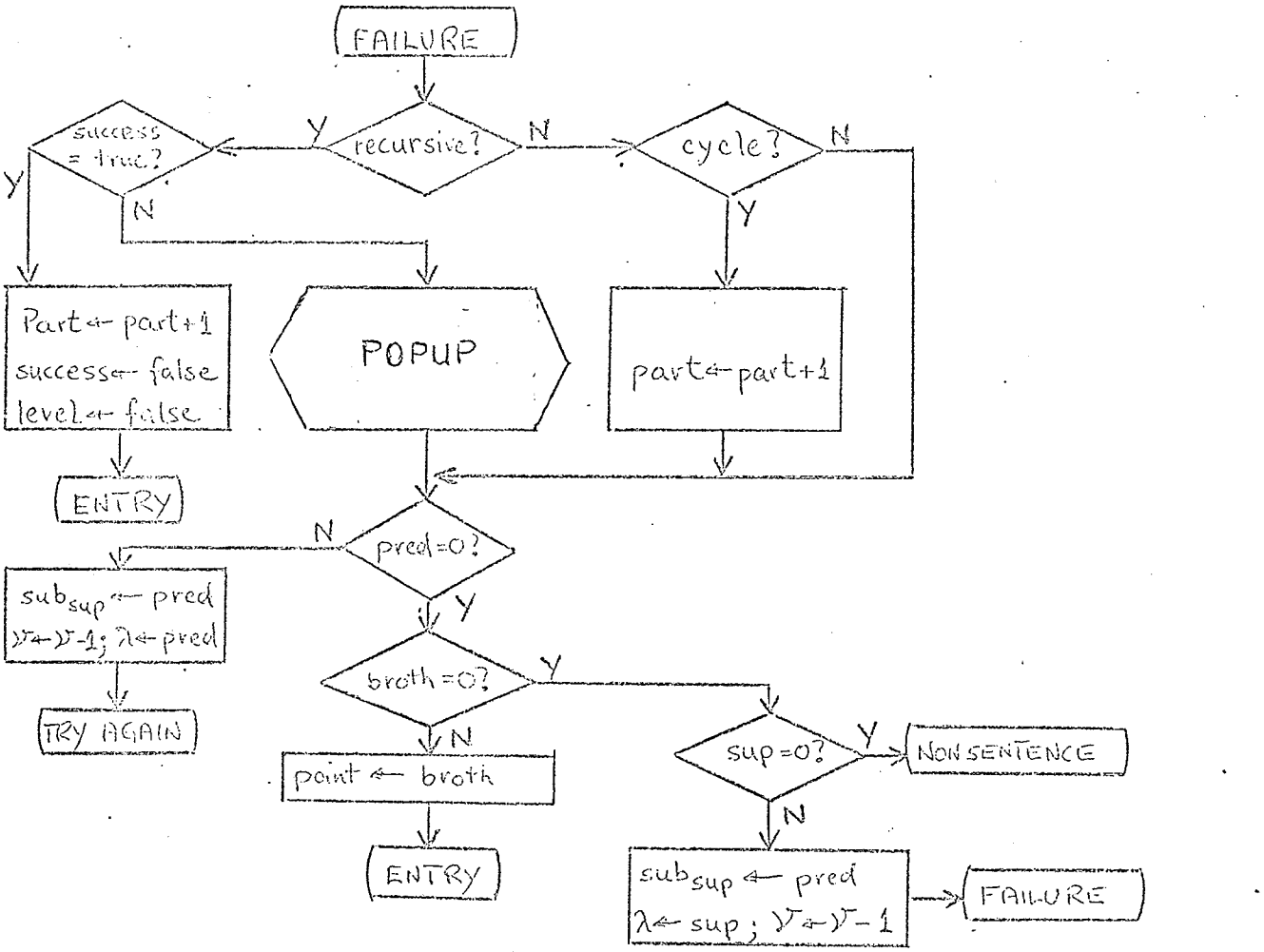
to point to the previous recursion list line which was created by the recursive node A in "w_1,A,w_2".

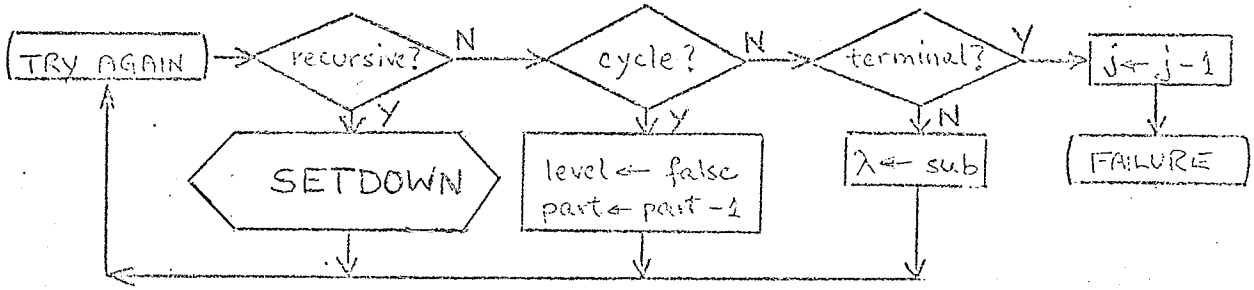
If subsequent "try again" commands are given, then the recursion list lines will be reestablished in exactly the opposite order in which they were created; an order which exactly parallels the order in which the goals of the parser are reestablished. It is obvious that the four routines "pushdown", "popup", "setdown", and "setup" are exactly what is needed to ensure that the correct recursion list line is always available.

A GENERAL PARSING ALGORITHM

FOR CONTEXT - FREE GRAMMARS




FAILURE
recursive?
Y
N
success = true?
Y
N
cycle?
N
Y
Part ← part+1
success ← false
level ← false
POPUP
part ← part+1
ENTRY
pred=0?
N
Y
sub_sup ← pred
λ ← pred
broth=0?
Y
N
TRY AGAIN
point ← broth
ENTRY
sup=0?
Y
NON SENTENCE
N
sub_sup ← pred
λ ← sup ; ν ← ν-1
FAILURE


TRY AGAIN
recursive?
N
Y
cycle?
N
Y
terminal?
Y
N
j ← j-1
SETDOWN
level ← false
part ← part-1
λ ← sub
FAILURE

Chapter 5

PARSING OF ALGOL-68

In this chapter the two-level syntax notation used in Algol-68 (4) will be explained. This will be followed by a description of how two-level syntax may be partially transformed into one-level syntax and what the limitations of this process are. Occasionally, production rules of Algol-68 will be referred to by their number in the report(4). The notation used will be (#rule number). Thus, (#8.2.2.a) refers to rule number "8.2.2.a" of the Algol-68 report (4).

5.1 BASIC DEFINITIONS

Recall from chapter 1 how the syntax of a context-free grammar was expressed. There was a finite alphabet $\underline{A} = \underline{S} \cup \underline{D}$ where

$\underline{S} = \{a, b, c, \ldots, z\}$ was the set of "syntactic

marks" or "small syntactic marks", and

$\underline{D}$= {:, ;, .., ,} was the set of "other syntactic marks".

In the case of two-level syntax, $\underline{A}$ is extended to $\underline{A}=\underline{S} \cup \underline{D} \cup \underline{L}$, where $\underline{L}$= {A, B, C, ..., Z} is known as the set of "large syntactic marks".

Now define a "metanotion" as a nonempty sequence of large syntactic marks (recall that a "notion" was a sequence of small syntactic marks).

Two-level syntax may now be said to consist of three quite distinct components:

1. an "underlying context-free grammar" and

2. a "superimposed context-free grammar" and

3. a "substitution mechanism".

5.1.1 The underlying context-free grammar

The underlying context-free grammar is a c.f.p.s.g. in the usual sense as defined previously with the exception that notions may consist of sequences of both "small syntactic marks" and "large syntactic marks". In other words, notions may have embedded within them one or more metanotions. Thus, "virtual MODE declarer: virtual MODE declarator; MODE mode indication." would be a valid production rule of the underlying context-free grammar.

Two other terms that will be used in connection with the production rules of the underlying context-free grammar must be defined.

Definition 5.1

A metanotion is termed a "free metanotion of a production rule" if it occurs only in the r.h.s. of the production rule and not in the l.h.s. of the rule.

Definition 5.2

A metanotion is termed a "constrained metanotion

of a production rule" if it occurs in the l.h.s. of the production rule.

5.1.2 The superimposed context-free grammar

The superimposed context-free grammar is a c.f.p.s.g. defined as follows:

$G=(\underline{V}_N, \underline{V}_T, \underline{P}, Z)$ where $\underline{V}_N = \{x \mid x$ is a metanotion$\}$,

$\underline{V}_T = \{y \mid y$ is a sequence of small syntactic marks$\}$

$\underline{P} = \{r \mid r=x:z.$ where $x \in \underline{V}_N, z \in (\underline{V}_N \cup \underline{V}_T)^*\}$,

and Z is any $x \in \underline{V}_N$. Z will in fact usually be a different metanotion every time the superimposed grammar is used. The function of this grammar consists of returning a string of terminals produced for any particular metanotion Z upon request, where Z is a metanotion that is supplied at the time of request and may be a different metanotion for each request.

It is further stipulated that any subset Q of P where Q $= \{x{:}z_1., x{:}z_2. , \ldots, x{:}z_n.\}$ may be replaced by "$x{:}z_1; z_2; \ldots ; z_n.$".

This definition of the superimposed context-free grammar is in fact very similar to the definition of context-free grammars given in chapter 1. The following differences should be noted:

1. whereas a (terminal) "symbol" of a context-free grammar is a notion ending with the protonotion "symbol", a (terminal) "symbol" of the superimposed context-free grammar is a sequence of small syntactic marks.

2. whereas a "non-terminal" of a context-free grammar is a notion consisting of small syntactic marks, a "non-terminal" of the superimposed context-free grammar is a metanotion.

3. sequences of notions in production rules of a context-free grammar are separated by commas to

make possible the identification of the end of one notion and the start of the next. In the superimposed context-free grammar, blanks serve to separate the terminals and non-terminals. Blanks embedded within a sequence of small syntactic marks do not take part in this rule as they are considered as belonging to the sequence of small syntactic marks. Thus, in "VICTAL reference to MODE declarator" appearing on the r.h.s. of some production rule of the superimposed context-free grammar, there are four separate components:

a) a non-terminal "VICTAL" followed by

b) a terminal "reference to" followed by

c) a nonterminal "MODE" followed by

d) a terminal "declarator".

4. The superimposed context-free grammar does not have a single head in the sense of a normal context-free grammar.

Note that it is impossible to have two terminals occuring together in a rule of the superimposed context-free grammar since any sequence of small syntactic marks is considered as a single terminal.

However, non-terminals of the superimposed context-free grammar may occur together and are separated by a blank when they do.

5.1.3 The substitution mechanism

We now consider a two-level grammar as a generator of sentences. This will be done by producing sentential forms consisting of (terminal) symbols from the head of the language. This head will be some non-terminal of the underlying context-free grammar. The productions are themselves taken only from the underlying context-free grammar. If, during the application of a production rule, free metanotions occur, then we invoke a mechanism which, using the superimposed context-free grammar, will deliver a sequence of small syntactic marks generated fron the free metanotion of the underlying context-free grammar. The production rule of the underlying context-free grammar is then applied in order to produce another sentential form. Furthermore, if the free metanotion occurs more than once in the production rule, it must be substituted for consistently throughout that production rule.

The situation may now arise that the mechanism described will generate a notion (consisting of course of small syntactic marks) which does not appear as the l.h.s. of any production rule of the underlying context-free grammar. However, there may be production rules with a l.h.s. containing constrained metanotions, such that, when these constrained metanotions are replaced by small syntactic marks (obtained according to the production rules of the superimposed context-free grammar), then the l.h.s. so obtained is identical to the notion of small syntactic marks indicated above.

For example, suppose the superimposed context-free grammar contains the rule "MODE: real; integer." and the underlying context-free grammar contains the rule "reference to MODE: reference symbol, MODE symbol." but does not contain a rule having a l.h.s. of "reference to real.". Assume now that for some sentential form the notion "reference to real" was generated. There is no rule of the underlying context-free grammar having this notion as a l.h.s. but

"reference to real" may be obtained from "reference to MODE" by substituting "real" for the constrained metanotion "MODE" where "real" was produced from "MODE" using the superimposed context-free grammar.

In such a case, the rule with the l.h.s. "reference to MODE" may be applied to produce the next sentential form providing the following condition is met:

> all occurrences of constrained metanotions within the production rule being applied are first replaced by the sequences of small syntactic marks which were generated from the constrained metanotions. This substitution must be consistent throughout the entire production rule. Thus, in the previous example, the production rule that would be applied is not "reference to real: reference symbol, MODE symbol." but is instead "reference to real: reference symbol, real symbol.".

This completely defines the operation of two-level syntax insofar as generation of sentences is concerned. These operations may of course also form

the basis of a recognizer of sentences.

Several problems have been conveniently ignored in this discussion such as for example which production rule of the underlying context-free grammar is a possible candidate for a production to be applied to some notion. These problems are however of no further importance here. The interested reader is referred to the Master's thesis of B. Wiebe who is studying these problems of two-level syntax analysis(5).

We may consider two-level syntax as a device described by a finite number of two-level syntax rules. This device has the capability of generating an infinite number of one-level syntax rules. We may also say that this two-level syntax device defines an infinite number of context-free grammars.

For any particular program, only one of these infinite number of context-free grammars which describe that program need be considered. However, this context-free grammar must first be generated by the two-level syntax device. This process may involve the

generation of a large number of context-free grammars before a suitable grammar describing the program is found.

Thus, the two-level syntax may be considered as being a generator of one-level syntax, context-free grammars which describe certain sentences being parsed. This mechanism makes it relatively easy to enforce context dependency such as for example restricting the use of real identifiers to those that have been previously declared as real identifiers. This context dependency may be completely described syntactically using two-level syntax. The following grammar demonstrates this property.

Example 5.1

a) Superimposed context-free grammar

1. COUNT: x; COUNT x.

2. LETTER: a; b; c.

b) Underlying context-free grammar

I) program: COUNT a, COUNT b, COUNT c.

II) COUNT x LETTER: x LETTER, COUNT LETTER.

III) x LETTER: LETTER symbol.

The head of the grammar is the notion "program".

This two-level syntax grammar now defines sentences which are of the form "a^n b^n c^n ". For example, "aabbcc" is generated as follows:

program $\Longrightarrow$ xxa, xxb, xxc. (since COUNT $\overset{*}{\Longrightarrow}$ xx by 1.) by application of I).

For each of xxa, xxb, and xxc apply rule II) with COUNT= x and LETTER= respectively a, b, and c. Thus, for xxa, the production rule "II) xxa: xa, xa." is applied.

Application of III) will then yield "xa: asymbol.", since "LETTER $\Longrightarrow$ a" by 2. .

There is however one serious difficulty with the two-level generator as it has been described. A large amount of time may conceivably be spent in the generation of one-level syntax grammars which must then

be followed by the parsing mechanism for parsing the input sentence according to the one-level syntax grammar generated. Failure would then cause the two-level syntax mechanism to generate new one-level grammars which will be used to parse the input sentence. It is evident that this whole mechanism will tend towards a rather timeconsuming parsing process and it would be preferable if some device for increasing the parsing efficiency could be employed.

An alternative approach would be to attempt the reduction of a two-level grammar to an equivalent one-level grammar. Such a process and its limitations are discussed next.

5.2 REDUCTION OF TWO-LEVEL SYNTAX TO ONE-LEVEL SYNTAX

Let us take for example the Algol-68 production rule (#2.1.d)"particular program: label sequence option, strong CLOSED void clause.". The production rule for "CLOSED" in the superimposed context-free grammar of Algol-68 states that "CLOSED: closed;

collateral; conditional."(#1.2.3.d). We now rewrite the rule for "particular program" as "particular program: label sequence option, first new notion." and add to the underlying context-free grammar for Algol-68 the new rule "first new notion: strong closed void clause; strong collateral void clause; strong conditional void clause.". The language generated by this new grammar is of course exactly the same as the language generated by the old grammar. Notice however that the metanotion "CLOSED" has been removed from this particular set of production rules and that what remains are two one-level syntax rules. Furthermore, there need not be any difficulty with semantic actions associated with the new production rules since all information that existed in the two-level syntax may still be deduced from the corresponding one-level syntax.

The mechanism just explained will work for all metanotions which produce a finite number of sequences of small syntactic marks. For example, metanotions such as "CLOSED", "ADJUSTED", "THELSE", "NINE" of Algol-68 are such metanotions(#1.2). In fact, a major proportion of metanotions of Algol-68 are of this kind.

5.2.1 "finite" and "infinite" metanotions

The metanotions producing a finite number of sequences will be referred to simply as "finite metanotions". Metanotions from which an infinite number of sequences of small syntactic marks may be generated will be termed "infinite metanotions." "COUNT" of example 5.1 was such an "infinite metanotion."

The property that makes metanotions infinite is of course the fact that productions of the form "$M_1 \overset{*}{\Longrightarrow} w_1$, M_2, w_2." $\overset{*}{\Longrightarrow} w_1$, w_3, M_2, w_4, w_2.", (where $M_2 \Longrightarrow w_3$, M_2, w_4.) are possible. In other words, some form of recursion exists (in this case $M_2 \overset{*}{\Longrightarrow} w_3$, M_2, w_4.). For example, one such infinite metanotion in Algol-68 is "MODE" which amongst "real", "integer", and others also produces "reference to MODE". Thus, "MODE" may produce for example "(reference to)^n real" where $n \geq 0$ and may therefore clearly produce an infinite number of sequences of small syntactic marks. Some other such infinite metanotions in Algol-68 are "LONGSETY",

"NOTION", and other metanotions which are related to MODE. At this point, it is instructive to consider a set of production rules from the Algol-68 report.

Example 5.2

a) superimposed context-free grammar

The grammar contains rules which make it possible to derive "real" and also "reference to MODE" from the metanotion "MODE" (#1.2.1.a).

b) underlying context-free grammar.

The grammar contains amongst others two rules similar to

1. (#7.1.1.b) virtual MODE declarer: virtual MODE declarator.

2. (#7.1.1.1) virtual reference to MODE declarator: reference to symbol, virtual MODE declarer.

Rule 1. may be changed to read "virtual MODE declarer: virtual real declarator; ... other 'virtual MODE declarators' with some substitution for MODE" ...; virtual reference to MODE declarator.". Here we have merely substituted for "MODE" some of the things that "MODE" produces. In the last alternative above, "reference to MODE" was substituted for "MODE". Rule 2. now defines a "virtual reference to MODE declarator" and application of the production rule during a parse will then make a "reference to symbol" a goal and will subsequently require the recognition of a "virtual MODE declarer." Note however that the rule 1. has been altered and does indeed already define completely what a "virtual MODE declarer" is. It is obvious that this mechanism has again produced one-level syntax rules which are equivalent to the previous two-level syntax rules. (One may for example substitute "mode" for "MODE" and the grammar would remain the same in this particular instance.) Surprisingly enough, this mechanism again works for the majority of infinite metanotions in Algol-68. In fact, a thorough study of the 288 production rules of Algol-68 revealed only two

production rules containing infinite metanotions for which this process does not work (#8.2.1.1.a) and (#8.2.2.1.a) . It is possible that there may be one or two more cases where the mechanism will not work and that these were not found in the preliminary investigation because of the complexity of the grammar. Any such cases will be discovered later when a complete reduction of the two-level syntax of Algol-68 to one-level syntax will be completed. At any rate, it may be safely stated that the percentage of rule of Algol-68 which may be transformed from two-level syntax to one-level syntax is approximately 99%. This transformation however does create additional problems which will be covered shortly (see 5.3).

The infinite metanotions for which the above procedure does not work are of the following kind:

1. a metanotion "A" is defined as "A: a; bA.", and

2. there exists a production rule "cA:x;cbA." in the underlying context-free grammar.

The problem in this case occurs because we may have "$cA \Longrightarrow cbA \Longrightarrow cbbA \Longrightarrow \ldots \Longrightarrow cb^nA \Longrightarrow \ldots$" and no finite number of one-level syntax rules can describe this infinite set of notions "cb^nA". Fortunately, as has been mentioned, only two of these rules (#8.2.1.1.a) and 8.2.2.1.a) have been found in the syntax of Algol-68. These rules may be circumvented as will be indicated later.

5.3 THE CONTEXT PROBLEM

One serious problem arises when two-level syntax rules are changed to one-level syntax rules. If a metanotion occurs more than once in some alternative of a production rule, the two-level syntax demands that it be substituted for consistently. It is this device which assured that the grammar of Example 5.1 generated only sentences of the form "$a^n\ b^n\ c^n$ ". During translation from two-level syntax to one-level syntax, this context dependency is lost. Example 5.3 demonstrates how the two-level grammar of Example 5.1

is changed to a one-level grammar with this context problem.

Example 5.3

1. program: COUNT a, COUNT b, COUNT c.
2. COUNT a: xa; COUNT x a.
3. COUNT b: xb; COUNT x b.
4. COUNT c: xc; COUNT x c.
5. xa: asymbol.
6. xb: bsymbol.
7. xc: csymbol.
8. COUNT x a: xa, COUNT a.
9. COUNT x b: xb, COUNT b.
10. COUNT x c: xc, COUNT c.

This grammar is a one-level syntax grammar since no special significance is attached to "COUNT"; in fact, it may be replaced by "count". It is obvious though that this grammar will not only generate "a^n b^n c^n" but will generate sentences of the form "a^x b^y c^z" for any $x,y,z \geq 1$. This is so because rule 1. as a two-level syntax rule ensured that the number of a's, b's, and c's generated will always be equal. However,

this condition is not met by the one-level grammar above.

At first glance it would appear that this is a serious problem in reducing two-level syntax to one-level syntax. However, an examination of the 288 production rules of Algol-68 reveals that only 19 rules will give rise to this problem. A closer examination of these rules shows them to be generally in one of two broad categories:

1. One category of rules ensures that certain classes of used identifiers are identical to those that have previously been declared to be in that class of identifiers, and

2. the second class of rules concerns itself with changes of type (or "mode" as it is called in Algol-68) in say an assignment statement.

Traditionally, the use of identifiers has not been connected by syntax to their declaration. Instead, tables are generally built up for identifiers

and there is no reason why the traditional approach in this case should not be used for Algol-68. We may thus assume that rules giving context problems insofar as declaration and use of identifiers is concerned, may be ignored in the reduction of two-level syntax to one-level syntax and that traditional methods of handling this problem exist.

The second instance where this context problem occurs is not so easily disposed of. Other languages such as Algol-60 have cases where type changes occur such as for example the assignation of an "integer" to a "real". However, in these other languages, the basic types or modes have generally been finite whereas Algol-68 allows an infinite number of modes. An infinite variety of type changes must therefore be allowed for in Algol-68. Thus, the way in which modes are changed from some a priori mode to a mode required in some certain context has been rigidly defined by two-level syntax in Algol-68. These changes of mode are known as "coercions" in Algol-68. Reduction of the two-level syntax to one-level syntax causes the information of what coercions are necessary in any one

case to be lost.

One possible solution to this problem would be to completely remove the rules governing coercions from the syntax of Algol-68 and shift the burden of determing the type of coercions required to the semantic routines. At any rate, the one-level syntax derived from the two-level syntax will always be able to pass on to the semantic routine the a priori mode found in some context and the a posteriori mode that is required in that context. The semantic routines will then have to ensure that the coercions are carried out properly.

5.3.1 Rewriting Algol-68 syntax

It must be obvious at this stage that rewriting the two-level syntax of Algol-68 will result in a one-level syntax of Algol-68 containing a sizeable number of production rules. A partial rewriting of Algol-68 as a one-level grammar done by the writer has resulted in 800 one-level syntax rules with the end of the translation not yet in sight. A reasonable estimate would place the total number of one-level

syntax rules for Algol-68 at about 1500. It must also be appreciated that the one-level grammar generated will by no means be a final one but may be expected to undergo considerable change during the writing of an Algol-68 compiler. Difficulties with context problems previously discussed should make it clear that changes to the syntax should be expected.

As anyone that has worked with restricted grammars (such as for example the Wirth-Weber precedence grammars previously mentioned) knows, changes to a grammar will generally cause disturbances in other parts of the grammar and can often require a fair amount of work in "fixing up" the grammar to conform to given restrictions. With a large grammar of about 1500 production rules, the amount of "grammar fixup" time and effort may well become prohibitive.

5.4 RELEVANCE OF THE GENERAL PARSING ALGORITHM

The need or usefulness of a general parsing algorithm that was developed in the previous chapters

is of course apparent now. The reduction of Algol-68 into one-level syntax will generate well in excess of a thousand production rules which may be expected to undergo changes during the initial compiler-writing attempts.

In order that efforts may be concentrated on the semantic actions and administration of data areas during the initial compiler-writing stages, it will be necessary to keep grammar "fixups" to a reasonable minimum. Once the semantic routines and representations of data areas have been "debugged" and "frozen" for some compiler, efforts may then be concentrated on the comparatively simple problem of designing an efficient parsing method for the compiler.

The general parsing algorithm developed in this thesis fulfills these requirements since it will accept any context-free grammar. It has the added advantage that syntax rules do not have to be transformed and this will make the "debugging" of the semantic routines somewhat easier.

Chapter 6

CONCLUSION

In conclusion, the general parsing algorithm presented here is not very suitable to production compilers. Due to its generality and backtracking ability, it entails a certain amount of inefficiency which should not be built into a production compiler if it can be avoided. However, it is a very useful algorithm for the development of a compiler since it requires little effort to be expended on the parsing aspect of the compiler and allows the major effort to be directed to solving the semantic aspects of the compiler.

A second topic touched on in chapter 5 was the reduction of the two-level syntax of Algol-68 to one-level syntax. It can be concluded that it is generally not possible to reduce two-level syntax to one-level syntax. However, if one restricts oneself to actual two-level grammars used in describing

programming languages (specifically Algol-68), then the problem of reducing two-level syntax to one-level syntax is generally solvable with only a few difficulties. The conclusion is that the reduction of the grammar of Algol-68 to one-level syntax is a promising exercise and should be carried out.

However, when this grammar transformation is applied to the syntax of Algol-68, a rather large context-free grammar estimated at about 1500 production rules results. Furthermore, this grammar may be expected to undergo considerable modification during the compiler-writing process. This necessitates the existence of as general a parsing algorithm as possible in order that the main efforts of the compiler writing process may be concentrated on the semantic aspects of the compiler. Once the semantic problems have been solved, efforts can be concentrated on providing an efficient parsing algorithm based on the grammar finally obtained.

Chapter 5 in fact demonstrates the utility of the general parsing algorithm developed in this thesis

and constitutes the major motivation for the development of this algorithm.

Aside from this, no general top-down, left-right parsing algorithm for context-free grammars was known to the writer and the development of such an algorithm is thus a valid academic exercise.

It is appropriate at this point to mention briefly some work done on parsing of LR(k) grammars. A knowledge by the reader as to the nature of LR(k) grammars is assumed here. The reader not familiar with LR(k) grammars is referred to (6) for a description of them.

Two algorithms for parsing LR(k) grammars which may be considered as being representative of the work on LR(k) grammars are discussed here. The first is basically a bottom-up algorithm by De Remer (9) and the second is a top-down algorithm by Earley (10). Both are quite general in their ability to handle LR(k) grammars which form a large subset of the context-free grammars. Indeed, the bulk of the grammars describing existing

languages are essentially LR(k). De Remer in fact reasons in (9) that a language designer will normally design LR(k) grammars quite naturally and the languages developed to date appear to support this contention.

Both algorithms, however, are not completely general because k must be specified. De Remer in (9) sets a maximum value on k, (usually k=3) and then proceeds to accept a grammar. His algorithm initially assumes k to be zero and develops a simple and efficient parser for the grammar. If it is found that the grammar is not LR(0), k is increased and so is the complexity of the parser generated. This process continues up to the maximum value of k. If at that point, the grammar is still not LR(k), it is assumed that the grammar is ambiguous and the algorithm stops (see for example (7) where it is shown that LR(k) grammars are unambiguous).

Earley's algorithm carries out a similar process although in his algorithm k appears to be fixed at its maximum value. It must therefore be ensured that any given grammar is indeed LR(k) before his algorithm is

used.

We thus see that both these algorithms are not general for context-free grammars since they are restricted to LR(k) grammars where some value for k must be specified. A further complication that arises is that the demands of the algorithms on computer resources (time and space) increase with k and that these will place some finite upper limit on k.

The algorithm developed in this thesis however is perfectly general for all context-free grammars and indeed will accept an LR(k) grammar where $k \rightarrow \infty$. The algorithm is furthermore independent of k.

It should of course be noted that the algorithms due to De Remer and Earley are very efficient compared to the algorithm presented here, especially for LR(k) grammars with a low value for k. However, their comparative efficiency diminishes as k increases. A finite upper bound for k exists in both De Remer's and Earley's algorithm if these algorithms are to be implemented on modern computers. This difficulty does not exist with our algorithm.

SUGGESTIONS FOR FUTURE WORK

Since this thesis arose out of a short preliminary investigation of Algol-68 and its compilation, the scope for future work remains large.

It is suggested that the general parsing algorithm developed here be made more efficient by for example rewriting it in a low level language. Several features may also be added to the parsing algorithm to improve its efficiency. One such feature would be to prevent backtracking generally and allow backtracking only when it is specifically allowed for some notion. The algorithm is sufficiently open-ended to allow this to be done with ease by inserting special codes into the code field of the syntax definitions and providing a minor alteration to take non-backtracking into account.

There should also be routines that translate syntax rules from some form easily readable to the user to the syntax table which is used internally. This is essentially a trivial problem.

Another more involved, but possible, routine that should be developed is one which would use the syntax table generated in order to identify cycles, recursive nodes, and cycle nodes since this process may take a considerable amount of time manually for involved grammars.

The major project that must be performed, and it is hoped that this will be completed by the summer of 1971, is the rewriting of Algol-68 as a one-level syntax and a thorough examination of the grammar thus obtained.

APPENDIX

This appendix consists of a listing of a PL/1 program for the general parsing algorithm plus a few examples demonstrating its operation. The program listed here was written in order to test the general parsing algorithm and as such contains many statements that were used to provide trace output so that the operation of the program could be easily monitored. A further design philosophy was to make it as easy as possible to debug the program through the use of modular design.

The program listed here is thus not intended to be an efficient encoding of the general parsing algorithm but is merely a test program. It has been included in this thesis along with a few examples in order to demonstrate the operation of the general parsing algorithm with actual examples. It should also be noted that some minor differences exist between the program and the algorithm developed in the text of the thesis.

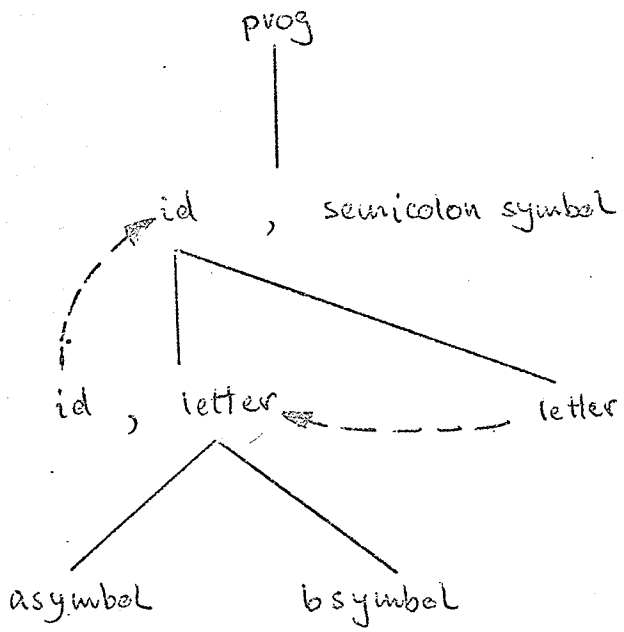
Example 1

prog : id, semicolon symbol.

id : id, letter; letter.

letter : asymbol; bsymbol.

Definition tree



Example 2

prog: a0, semicolon symbol.

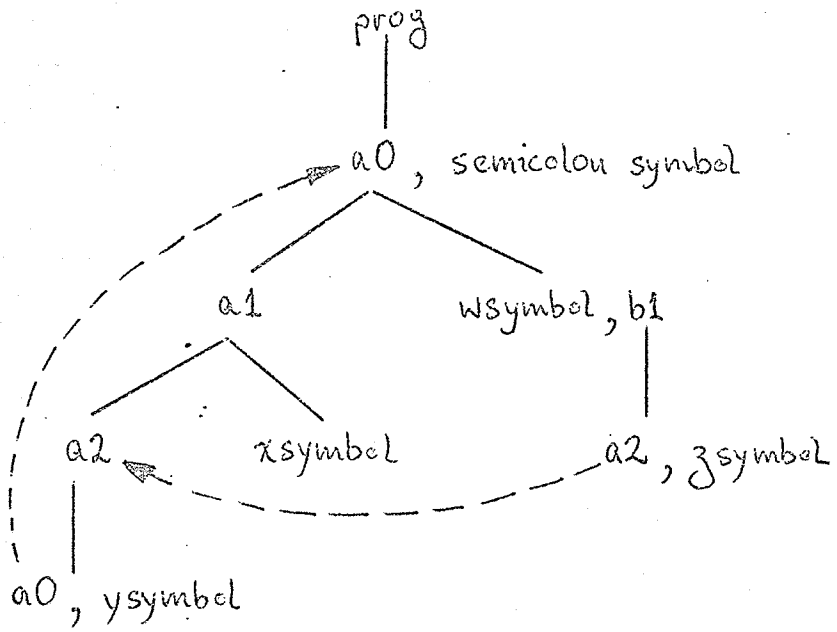
a0: a1; wsymbol, b1.

a1: a2; xsymbol.

a2:a0, ysymbol.

b1: a2, zsymbol.

Definition tree



Example 3

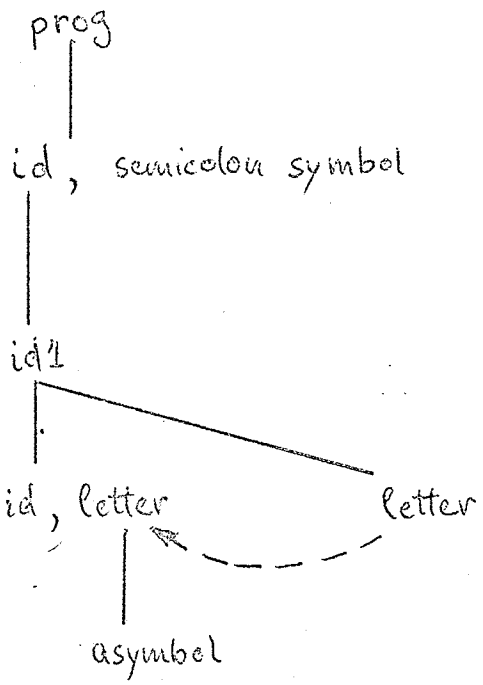
prog: id, semicolon symbol.

id : id1.

id1 : id, letter; letter.

letter: asymbol.

Definition tree



Example 4

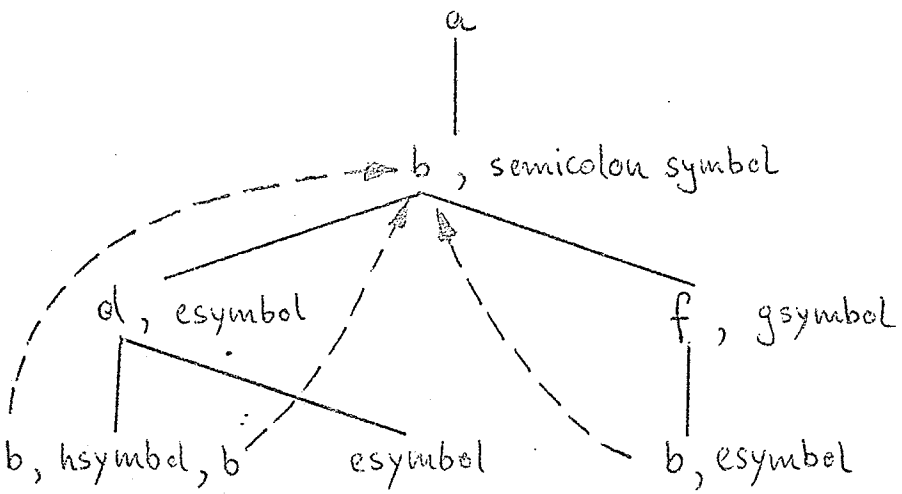
a : b, semicolon symbol.

b : d, esymbol; f, gsymbol.

d : b, hsymbol, b; esymbol.

f : b, esymbol.

Definition tree



Example 5

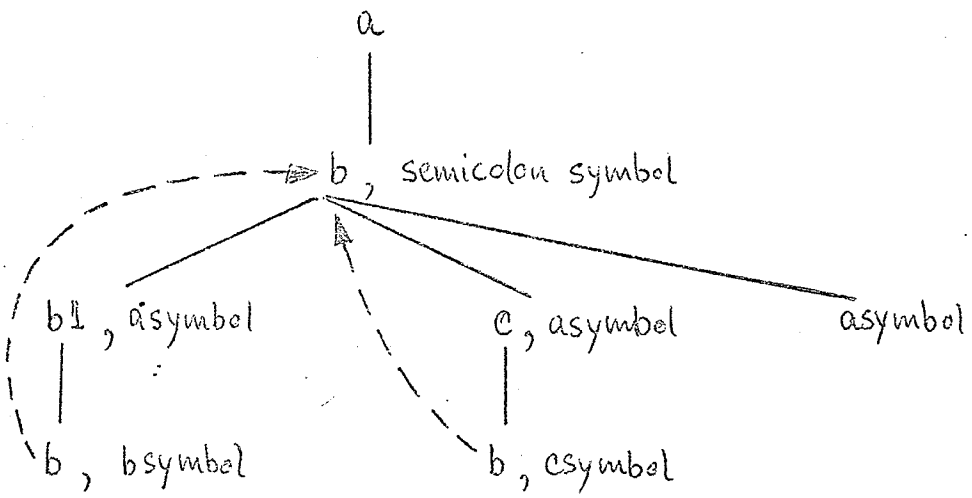
a : b, semicolon symbol.

b : b1, asymbol; c, asymbol; asymbol.

b1 : b, bsymbol.

c : b, csymbol.

Definition tree



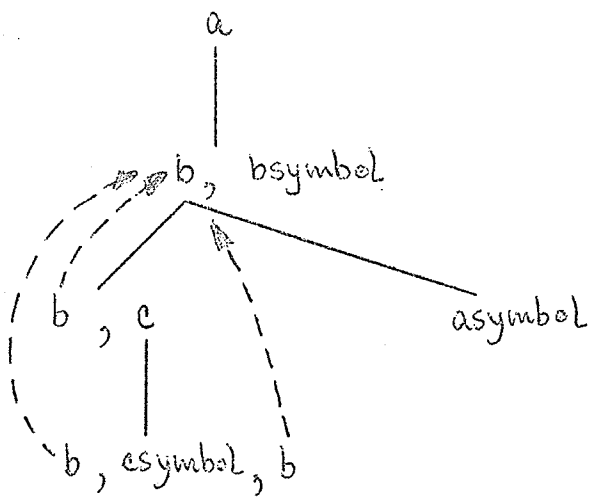
Example 6

a : b, bsymbol.

b : b, c; asymbol.

c : b, csymbol, b.

Definition tree



Example 7

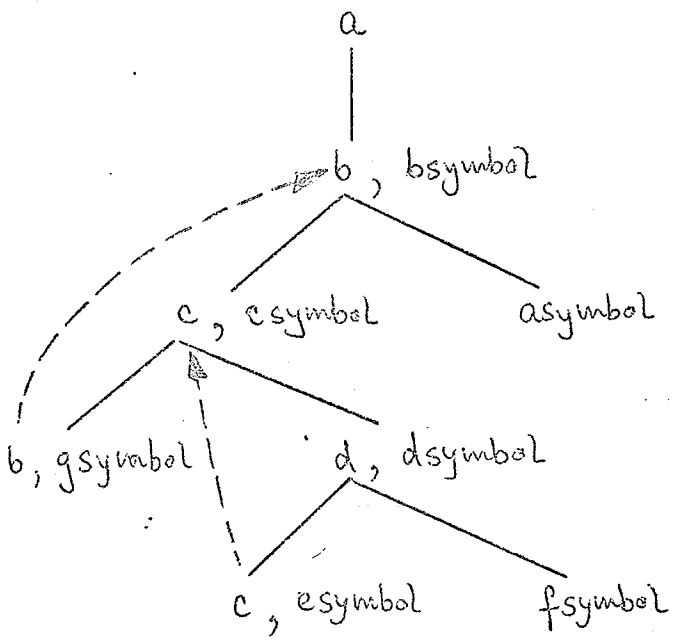
a : b, bsymbol.

b : c, csymbol; asymbol.

c : b, gsymbol; d, dsymbol.

d : c, esymbol; fsymbol.

Definition tree



```
1               MAIN: PROCEDURE (PARM) OPTIONS (MAIN);
2                 DCL PARM CHARACTER (100) VARYING;
3                    BEGIN;
4                 DCL (NGOAL, NRS, NRL, NAR, NTL, NSL, NSTRING) BIN FIXED;
5                 ON ENDFILE (SYSIN) BEGIN; GO TO STOP; END;
9               GET_DATA:
10                NGOAL=12; NRS=20; NRL=10; NAR=50; NTL=30; NSL=50; NSTRING=8000;
16                GET DATA;
17                BEGIN;

18                DCL (POINT, SUP, SUB, PRED, BROTHER, COMP, SON, CODE, ARP, ARPE,
                     RLP, RSP, RSPE, J, L, PART, SAVE, FINAL_SIZE) BIN FIXED,

                     (RS(NRS, 7), RL(NRL, 2), AR(NAR, 5), SL(NSL, 4)) BIN FIXED,      2 OUT

                     TL(NTL) CHARACTER (3) VARYING,
                     STRING CHARACTER (NSTRING) VARYING,
                     (COPY_1, OUTPUT) CHARACTER(100),

                     GOAL(0:NGOAL) LABEL,

                     (PUSHDOWN, POPUP, SETDOWN, SETUP, SET_AUX, PUT_RS) ENTRY,
                     P ENTRY (BIN FIXED) RETURNS (BIT(1)),
                     SET ENTRY (BIN FIXED, BIN FIXED, BIN FIXED, BIN FIXED),
                     PUT_AR ENTRY (BIN FIXED),
                     SEARCH ENTRY (BIN FIXED) RETURNS (BIN FIXED),
                     RESET ENTRY RETURNS (BIN FIXED),

                     (B, REC, CYC, ACT, SUCC, TERM, LEVEL) BIT(1);

19              P: PROCEDURE (I) RETURNS (BIT(1));
20                DCL I BIN FIXED;
21                IF SEARCH (I)=0
22                THEN RETURN ('0'B);
23                ELSE RETURN ('1'B);
24              END P;

25              POPUP: PROCEDURE;
26                BEGIN;
27                DCL (SUP, PRED) BIN FIXED;
28                SAVE=RSP;
29                SUP= RS(RSP,1); PRED=RS(RSP,3);
31                IF PRED=0 THEN RSP=SUP; ELSE RSP=PRED;
34                RS(SUP,2)= PRED;
35                RS(SAVE, 2)= RSPE;
36                RSPE= SAVE;
37                RL(RLP,2)= RSP;
38                IF B
39                THEN DO;
40                   CALL PUT_RS;
41                   PUT FILE (TRACE) EDIT('POPUP') (COLUMN(109), A);
42                   END;
43                END;
44              END POPUP;

45              PUSHDOWN: PROCEDURE;
46                SAVE=RS(RSPE,2);
```

```
47              RS(RSPE, 1)= RSP;   RS(RSPE, 3) = RS(RSP,2); RS(RSPE, 6) =1;
50              RS(RSPE,2), RS(RSPE, 4), RS(RSPE, 5) =0; RS(RSPE, 7)=J;          2 OCT
52              RS(RSP,2)= RSPE;
53              RL(RLP,2)= RSPE;
54              RSP=RSPE;
55              RSPE=SAVE;
56              IF 3
57              THEN DO;
58                 CALL PUT_RS;
59                 PUT FILE (TRACE) EDIT ('PUSHDOWN') (COLUMN(109), A);
60                 END;
61            END PUSHDOWN;

62            PUT_AR: PROCEDURE (ARP);
63              DCL ARP BIN FIXED;
64              PUT FILE (TRACE) EDIT
                (ARP, '|', AR(ARP,1), '|', AR(ARP,2), '|', AR(ARP,3), '|',
                   AR(ARP,4), '|') (COLUMN(1), 5 (F(5,0), A(1)));
65              PUT FILE (TRACE) EDIT ('|','|','|','|','|','|','|')
                   (COLUMN(61),(7) (X(5), A(1)));
66            END PUT_AR;

67            PUT_RS: PROCEDURE;
68              PUT FILE (TRACE) EDIT ('|','|','|','|','|')
                   (COLUMN(1), (5) (X(5), A(1)));
69              PUT FILE (TRACE) EDIT
                (RSP, '|', RS(RSP,1), '|', RS(RSP,2), '|', RS(RSP,3), '|',
                   RS(RSP,4), '|') (COLUMN(61), 5 (F(5,0), A(1)));
70              IF RS(RSP, 5) =1
71              THEN PUT FILE (TRACE) EDIT ('    T|') (A);
72              ELSE PUT FILE (TRACE) EDIT ('    F|') (A);
73              IF RS(RSP,6) =1
74              THEN PUT FILE (TRACE) EDIT ('    T|') (A);
75              ELSE PUT FILE (TRACE) EDIT ('    F|') (A);
76            END PUT_RS;

77            RESET: PROCEDURE RETURNS (BIN FIXED);
78              RETURN (J - LENGTH (TL(SON)));
79            END RESET;

80            SEARCH: PROCEDURE (A) RETURNS (BIN FIXED);
81              DCL A BIN FIXED;
82              BEGIN;
83              DCL (I,J,K,L) BIN FIXED;
84              IF NPL=0 THEN RETURN (0);
86              I=1; J=NPL; K=(I+J)/2; L=RL(K,1);
90              DO WHILE (L¬=A);
91                 IF L>A THEN J=K-1; ELSE I=K+1;
94                 K=(I+J)/2;
95                 IF L=RL(K,1) THEN RETURN (0); ELSE L=RL(K,1);
98                 END;
99              RETURN(K);
100             END;
101           END SEARCH;

102           SET: PROCEDURE (W, X, Y, Z);
103             DCL (W, X, Y, Z) BIN FIXED;
```

```
104               AR(ARP, 3) = ARPE;
105               AR(ARPE, 1) =W; AR(ARPE, 2)=X;
107               AR(ARPE, 3) =Y; AR(ARPE, 4) =Z;
109               AR(ARPE, 5)=J;
110               ARP=ARPE; ARPE=ARPE+1;
112               CALL SET_AUX;
113               IF B
114               THEN DO;
115                  IF SUP¬=0 THEN DO;
117                  CALL PUT_AR(SUP);
118                  PUT FILE (TRACE) EDIT('SUP UPDATE') (COLUMN(109), A);
119                  END;
120                  CALL PUT_AR(ARP);
121                  PUT FILE (TRACE) EDIT (ARP,ARPE)
                        (SKIP(0), COLUMN(37), (2)F(6,0));
122                  END;
123            END SET;

124            SET_AUX: PROCEDURE;
125              POINT = AR(ARP,1); SUP= AR(ARP,2);
127              SUB = AR(ARP,3); PRED = AR(ARP,4);

129              BROTHER = SL(POINT, 1); COMP=SL (POINT, 2);
131              SON= SL(POINT, 3); CODE=SL(POINT, 4);

                 /* CODES ARE: TERMINAL=10      */
                 /*            CYCLE=11         */
                 /*            RECURSIVE=12     */

133              IF CODE =10 THEN TERM='1'B; ELSE TERM='0'B;
136              IF CODE =11 THEN CYC='1'B; ELSE CYC= '0'B;
139              IF CODE=12 THEN REC='1'B; ELSE REC='0'B;

142              IF REC | CYC
143              THEN DO;
144                 RLP=SEARCH(SON);
145                 RSP=RL(RLP,2);
146                 IF RS(RSP,1)=0 THEN ACT='0'B; ELSE ACT= '1'B;
149                 IF ACT THEN IF RS(RSP,7)¬= AR(ARP,5) THEN ACT = '0'B;
152                 IF RS(RSP,5)=0 THEN SUCC='0'B; ELSE SUCC='1'B;
155                 IF RS(RSP,6)=0 THEN LEVEL='0'B; ELSE LEVEL='1'B;
158                 PART=RS(RSP,4);
159                 END;
160            END SET_AUX;

161            SETDOWN: PROCEDURE;
162              IF B THEN PUT FILE (TRACE) EDIT
                    ('|','|','|','|','|',  '|','|','|','|','|','|','|')
                    (COLUMN(1), (5) (X(5), A(1)), COLUMN(61), (7)(X(5), A(1)));
154              IF B THEN PUT FILE (TRACE) EDIT (RSP, ' SETDOWN TO', RS(RSP,2))
                    (COLUMN(109), F(4,0), A, F(4,0));
166              RSP=RS(RSP, 2);
167              RL(RLP, 2) = RSP;
168            END SETDOWN;

169            SETUP: PROCEDURE;
170              IF B THEN PUT FILE (TRACE) EDIT
```

```
                ('|','|','|','|','|',  '|','|','|','|','|','|','|')
                (COLUMN(1), (5)(X(5), A(1)), COLUMN(61), (7)(X(5), A(1)));
172             IF B THEN PUT FILE (TRACE) EDIT (RSP, ' SET UP TO', RS(PSP,1))
                (COLUMN(109), F(4,0), A, F(4,0));
174             RSP=RS(RSP,1);
175             RL(RLP,2) = RSP;
176           END SETUP;

177           INITIALIZATION:
                GET LIST ((I, SL(I,*) DO J=1 TO NSL));
178             PUT PAGE;
179             IF PARM='TRACE' THEN B='1'B; ELSE B='0'B;
192             IF B THEN OPEN FILE (TRACE) PRINT LINESIZE(130) PAGESIZE(60);
184             IF B THEN PUT FILE (TRACE) PAGE;
185             IF B
187             THEN DO;
188                PUT FILE (TRACE) EDIT ('ANALYSIS RECORD', 'RECURSION STACKS')
                   (X(7), A, COLUMN(74), A);
189                PUT FILE (TRACE) EDIT ('_____________________',
                   '_____________________________________')
                   (COLUMN(7), A(24), COLUMN(67), A(36));
190                PUT FILE (TRACE) EDIT
                   ('  NO.|PCINT|  SUP|  SUB| PRED|         APP  ARPE   ISP
                 NO.|  SUP|  SUB| PRED| PART| SUCC|LEVEL|') (COLUMN(1), A);
191                PUT FILE (TRACE) EDIT ('_____________________',
                   '_____________________________________')
                   (SKIP(1), COLUMN(7), A(24), COLUMN(67), A(36));
192                END;
193             PUT EDIT ('THE FOLLOWING ARRAY BOUNDS HAVE BEEN SPECIFIED OR ARE USED
                BY DEFAULT:') (A);
194             PUT DATA (NAP, NGOAL, NRL, NRS, NSL, NTL, NSTRING) SKIP;
195             PUT EDIT ('SYNTAX LIST', '   NO. BRCTH  COMP   SON  CODE')
                   (SKIP(3), A, SKIP, A);
196             PUT EDIT ((I, SL(I,*) DO I=1 TO NSL)) (SKIP, (5) F(6,0));
197             GET LIST (TL, STRING);
198             PUT EDIT ('TERMINAL LIST') (SKIP(2), A);
199             PUT EDIT ((I, TL(I) DO I=1 TO NTL)) (SKIP, F(6,0), X(1), A);
200             PUT EDIT ('INPUT STRING', STRING) (SKIP(2), A, SKIP, A);
201             J=1; NRL=0;

203             DO I=1 TO NSL;
204                IF SL(I,4)=11
205                THEN DO;
206                   IF ¬P(SL(I,3))
207                   THEN DO;
208                      RL(J,1)=SL(I,3);
209                      RL(J,2)=J;
210                      RS(J,*)=0;
211                      J=J+1; NRL=NRL+1;
213                      END;
214                   END;
215                END;

216             DO I=1 TO NRL;
217             DO K=1 TO NRL-I;

218             IF RL (K,1) > RL(K+1,1)
```

```
219                THEN DO L=1 TO 2;
220                   ITEMP=RL(K,L); RL(K,L)=RL(K+1,L); RL(K+1,L)=ITEMP;
223                   END;
224                END;
225                END;

226                PUT EDIT ('RECURSION LIST', (I,(RL(I,J) DO J=1 TO 2) DO I=1 TO NRL))
                      (SKIP(2), A,(NRL)(SKIP, (3) F(6,0)));

227                RSPE=NRL+1;
228                DO I=RSPE TO NRS-1;
229                   RS(I,2)=I+1;
230                   END;


231                GOAL(10) = GOAL_10;
232                GOAL(11), GOAL(12), GOAL(0) = GOAL_0;

233                ARPE=1; J=1; APP=1;
236                COPY_1= '     |     |     |     |     |                              |
                   |     |     |     |     |     |';
237                ON ENDPAGE (TRACE)
238                BEGIN;
239                   IF B THEN DO;
241                   PUT FILE (TRACE) EDIT ('_____________________________',
                      '_____________________________________')
                      (COLUMN(7), A(24), COLUMN(67), A(36));
242                   PUT FILE (TRACE) PAGE;
243                   PUT FILE (TRACE) EDIT ('_____________________________',
                      '_____________________________________')
                      (COLUMN(7), A(24), COLUMN(67), A(36));
244                   PUT FILE (TRACE) EDIT
                      ('  NO.|POINT|  SUP|  SUB| PRED|','   APP  ARPE     J',
                      '  NO.|  SUP|  SUB| PRED| PART| SUCC|LEVEL|')
                      (COLUMN(1), A, COLUMN(37), A, COLUMN(61), A);
245                   PUT FILE (TRACE) EDIT ('_____________________',
                      '_____________________________________')
                      (COLUMN(7), A(24), COLUMN(67), A(36));
246                   END; ELSE;
248                   END;
249                CALL SET(1,0,0,0);
250                GO TO ENTRY;

251                ENTRY:
252                  IF B THEN PUT FILE(TRACE) EDIT
                     ('|','|','|','|','|', '|','|','|','|','|','|','|','***ENTRY***')
                     (COLUMN(1), (5)(X(5), A(1)), COLUMN(61), (7)(X(5),A(1)),
                     COLUMN(109),A);
253                  IF REC
254                  THEN DO;
255                     CALL PUSHDOWN;
256                     GO TO GOAL(CODE);
257                     END;

258                  IF ¬CYC THEN GO TO GOAL(CODE);
260                  IF ¬ ACT THEN GO TO GOAL(CODE);
```

```
262                IF PART=0
263                THEN DO;
264                   IF BROTHER=0 THEN GO TO FAILURE;

266                   AR(ARP,1) = BROTHER;
267                   CALL SET_AUX;
268                   IF B
269                   THEN DO;
270                      CALL PUT_AR(ARP);
271                      PUT FILE (TRACE) EDIT ('BROTHER') (COLUMN(109), A);
272                      END;
273                   GO TO ENTRY;
274                   END;
275                ELSE DO ;
276                   RS(RSP,4) =PART-1;
277                   IF B THEN CALL PUT_RS;
279                   GO TO GOAL(CODE);
280                   END;

281             GOAL_10:
                  L=LENGTH (TL(SON));
282               L1:
                  IF SUBSTR(STRING, J, 1) = ' '
283               THEN DO;
284                   J=J+1;
285                   GO TO L1;
286                   END;

287               IF SUBSTR(STRING, J, L) = TL(SON)
288               THEN DO;
289                   J=J+L;
290                   IF B
291                   THEN DO;
292                      PUT FILE (TRACE) EDIT ('|', '|', '|', '|', '|', J, '|', '|',
                           '|', '|', '|', '|', '|', 'FOUND ', TL(SON))
                           (COLUMN(1), (5) (X(5), A(1)), COLUMN(49), F(6,0),
                           COLUMN(61), (7) (X(5), A(1)), COLUMN(109), A, A);
293                      END;
294                   FINAL_SIZE=ARP;
295                   GO TO SUCCESS;
296                   END;
297                ELSE DO;
298                   IF B THEN PUT FILE (TRACE) EDIT
                      ('|','|','|','|','|', '|','|','|','|','|','|','|')
                      (COLUMN(1), (5)(X(5),A(1)), COLUMN(61), (7)(X(5),A(1)));
300                   IF B THEN PUT FILE (TRACE) EDIT (TL(SON), ' NOT FOUND')
                         (COLUMN(109), A, A);
302                   GO TO FAILURE;
303                   END;

304             GOAL_0: GOAL_11: GOAL_12:
                  CALL SET (SON, ARP, 0, SUB);
305               GO TO ENTRY;

306             SUCCESS:
307               IF B THEN PUT FILE (TRACE) EDIT
                     ('|','|','|','|','|', '|','|','|','|','|','|','|','***SUCCESS***')
```

```
                  (COLUMN(1), (5)(X(5),A), COLUMN(61), (7)(X(5),A),COLUMN(109),A);
308               SAVE= COMP;
309               ARP=SUP;
310               CALL SET_AUX;
311               IF B
312               THEN DO;
313                  PUT FILE (TRACE) EDIT ('|', '|', '|', '|', '|', ARP, '|', '|',
                        '|', '|', '|', '|', '|', 'SUP')
                        (COLUMN(1), (5) (X(5), A(1)), COLUMN(37), F(6,0),
                        COLUMN(61), (7) (X(5), A(1)), COLUMN(109), A);
314                  END;
315               IF SAVE ¬=0
316               THEN DO;
317                  CALL SET (SAVE, ARP, 0, SUB);
318                  IF B THEN PUT FILE (TRACE) EDIT ('SUB_COMP') (COLUMN (109), A);
320                  GO TO ENTRY;
321                  END;

322               IF SUP=0 THEN GO TO SENTENCE;
324               IF REC
325               THEN IF LEVEL
326               THEN DO;
327                  RS(RSP,5) =1;
328                  IF B THEN CALL PUT_RS;
330                  CALL SETUP;
331                  GO TO SUCCESS;
332                  END;
333               ELSE DO;
334                  ARP=SUB;
335                  CALL SET_AUX;
336                  IF B
337                  THEN DO;
338                  PUT FILE (TRACE) EDIT ('|', '|', '|', '|', '|', ARP, '|', '|',
                        '|', '|', '|', '|', '|', 'SUB')
                        (COLUMN(1), (5) (X(5), A(1)), COLUMN(37), F(6,0),
                        COLUMN(61), (7) (X(5), A(1)), COLUMN(109), A);
339                     END;
340                  GO TO TRY_AGAIN;
341                  END;

342               IF ¬CYC THEN GO TO SUCCESS;
344               IF ¬ACT THEN GO TO SUCCESS;
346               IF PART=0 THEN RS(RSP,5)=1;
348               RS(RSP,4) =PART+1;
349               IF B THEN CALL PUT_RS;
351               GO TO SUCCESS;

352               FAILURE:
                  IF B THEN
353               PUT FILE (TRACE)EDIT ('|','|','|','|','|', '|','|','|','|','|','|',
                  '|','***FAILURE***') (COLUMN(1),(5)(X(5),A),COLUMN(61),(7)(X(5),A),
                  COLUMN(109),A);
354               IF REC
355               THEN DO ;
356                  IF SUCC
357                  THEN DO;
358                     RS(RSP,5)=0;
```

```
359                     RS(RSP,6)=0;
360                     RS(RSP,4) = PART+1;
361                     IF B THEN CALL PUT_RS;
363                     CALL SET (SCN, ARP, 0, SUB);
364                     IF B THEN PUT FILE (TRACE) EDIT ('SCN') (COLUMN(109), A);
366                     GO TO ENTRY;
367                     END;

368                  CALL POPUP;
369                  END;

370               IF PRED¬=0
371               THEN DO;
372                  AR(SUP,3)= PRED;
373                  IF B THEN DO;
375                     CALL PUT_AR(SUP);
376                     PUT FILE (TRACE) EDIT ('SUP UPDATE') (COLUMN(109), A);
377                     END;

378                  ARPE=ARPE-1;
379                  ARP=PRED;
380                  CALL SET_AUX;
381                  IF B
382                  THEN DO;
383                     CALL PUT_AR(ARP);
384                  PUT FILE (TRACE) EDIT ('|','|','|','|','|', ARP, ARPE,
                     '|','|','|','|','|','|','|','|')
                     (COLUMN(1), (5)(X(5),A), COLUMN(37), (2)F(5,0), COLUMN(61),
                     (7)(X(5),A));
385                     PUT FILE (TRACE) EDIT ('PRED') (COLUMN(109), A);
386                     END;

387                  GO TO TRY_AGAIN;
388                  END;

389               IF BROTHER¬=0
390               THEN DO;
391                  AR(ARP,1)= BROTHER;
392                  CALL SET_AUX;
393                  IF B
394                  THEN DO;
395                     CALL PUT_AR(ARP);
396                     PUT FILE (TRACE) EDIT ('BROTHER') (COLUMN(109), A);
397                     END;

398                   GO TO ENTRY;
399                  END;

400               IF SUP=0 THEN GO TO NONSENTENCE;
402               AR(SUP,3) =PRED;
403               ARP=SUP;
404               ARPE=ARPE-1;
405               CALL SET_AUX;
406               IF B
407               THEN DO;
408                  CALL PUT_AR(ARP);
409                  PUT FILE (TRACE) EDIT ('|', '|', '|', '|', '|', ARP, ARPE,'|','|',
```

```
                    '|', '|', '|', '|', '|', 'RESET TO SUP')
                    (COLUMN(1), (5) (X(5), A(1)), COLUMN(37), (2)F(6,0),
                    COLUMN(61), (7) (X(5), A(1)), COLUMN(109), A);
410                 END;

411               IF ¬CYC THEN GO TO FAILURE;
413               IF ¬ACT THEN GO TO FAILURE;
415               PART =PART+1;
416               PS(RSP,4)= PART;
417               IF B THEN CALL PUT_RS;
419               GO TO FAILURE;

420             TRY_AGAIN:
                  IF B THEN
421               PUT FILE (TRACE) EDIT ('|','|','|','|','|', '|','|','|','|','|','|',
                  '|','***TRY_AGAIN***') (COLUMN(1), (5)(X(5),A), COLUMN(61),
                  (7)(X(5),A), COLUMN(109), A);
422               IF TERM
423               THEN DO;
424                  J=RESET;
425                  IF B
426                  THEN DO;
427                     PUT FILE (TRACE) EDIT ('|', '|', '|', '|', '|', J, '|', '|',
                           '|', '|', '|', '|', '|')
                           (COLUMN(1), (5) (X(5), A(1)), COLUMN(49), F(6,0),
                           COLUMN(61), (7) (X(5), A(1)));
428                     END;
429                  GO TO FAILURE;
430                  END;

431               IF REC
432               THEN DO;
433                  CALL SETDOWN;
434                  IF B THEN CALL PUT_RS;
436                  END;
437               ELSE DO;
438                  IF CYC
439                  THEN IF ACT
440                  THEN DO;
441                     RS(RSP,6)=0;
442                     RS(RSP,4) = PART -1;
443                     IF B THEN CALL PUT_RS;
445                     END;
446                  END;
447               ARP=SUB;
448               CALL SET_AUX;
449               IF B
450               THEN DO;
451                  PUT FILE (TRACE) EDIT ('|', '|', '|', '|', '|', ARP, '|', '|',
                        '|', '|', '|', '|', '|', 'SUB')
                        (COLUMN(1), (5) (X(5), A(1)), COLUMN(37), F(6,0),
                        COLUMN(61), (7) (X(5), A(1)), COLUMN(109), A);
452                  END;
453               GO TO TRY_AGAIN;

454             SENTENCE:
                  PUT PAGE;
```

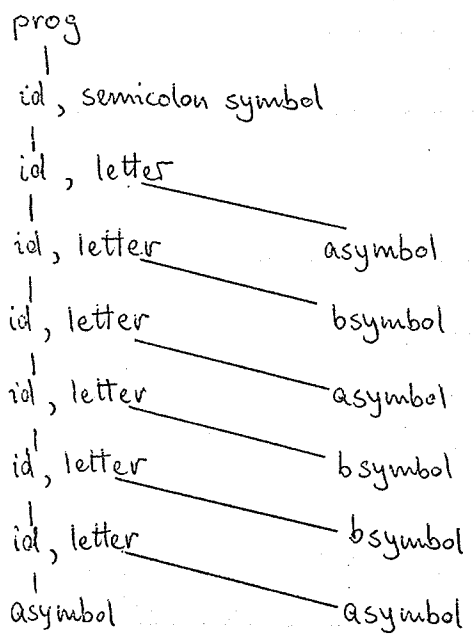
```
455                PUT EDIT ('SENTENCE WAS FOUND')(A);
456                GO TO L2;
457              NONSENTENCE:
                   PUT PAGE;
458                PUT EDIT ('NO SENTENCE WAS FOUND')(A);
459                GO TO L2;
460                L2:
                   PUT EDIT('THE FINAL ANALYSIS RECORD FOLLOWS')(SKIP, A);
461                PUT EDIT ('_________________________')(SKIP(2), X(6), A(24));
462                PUT EDIT('  NO.|POINT|  SUP|  SUB| PRED|')(SKIP, A);
463                PUT EDIT ('_________________________') (SKIP(0), X(6), A(24));
464                DO ARP=1 TO FINAL_SIZE;
465                   PUT EDIT (ARP,'|', AR(ARP,1), '|', AR(ARP,2), '|',
                        AR(ARP,3), '|', AR(ARP,4), '|')
                        (COLUMN(1), (5) (F(5,0), A(1)));
466                   END;
467                PUT EDIT ('_________________________') (SKIP(0),COLUMN(7), A);
468                GO TO GET_DATA;

469              STOP: END MAIN;
```

```
THE FOLLOWING ARRAY BOUNDS HAVE BEEN SPECIFIED OR ARE USED BY DEFAULT:
NAR=        50          NGOAL=        12          NRL=        10          NRS=        20          NSL=         9
NTL=         3          NSTRING=       0;


SYNTAX LIST
   NO.  BROTH  COMP   SON  CODE
     1      0     0     2     0
     2      0     8     3    12
     3      5     4     3    11
     4      5     0     6     0
     5      0     0     6     0
     6      7     0     1    10
     7      0     0     2    10
     8      0     0     3    10

TERMINAL LIST
     1 A
     2 B
     3 ;

INPUT STRING
AABBABA;

RECURSION LIST
     1     3     1
```

SENTENCE WAS FOUND
THE FINAL ANALYSIS RECORD FOLLOWS

NO.	PCINT	SUBI	SUBI	PREDI
1	1	0	23	0
2	2	1	21	0
3	3	2	19	0
4	3	3	17	0
5	3	4	15	0
6	3	5	13	0
7	3	6	11	0
8	3	7	9	0
9	5	8	10	0
10	6	9	0	0
11	4	7	12	3
12	6	11	0	0
13	4	6	14	7
14	7	13	0	0
15	4	5	16	6
16	7	15	0	0
17	4	4	18	5
18	6	17	0	0
19	4	3	20	4
20	7	19	0	0
21	4	2	22	3
22	6	21	0	0
23	8	1	0	2

ANALYSIS TREE

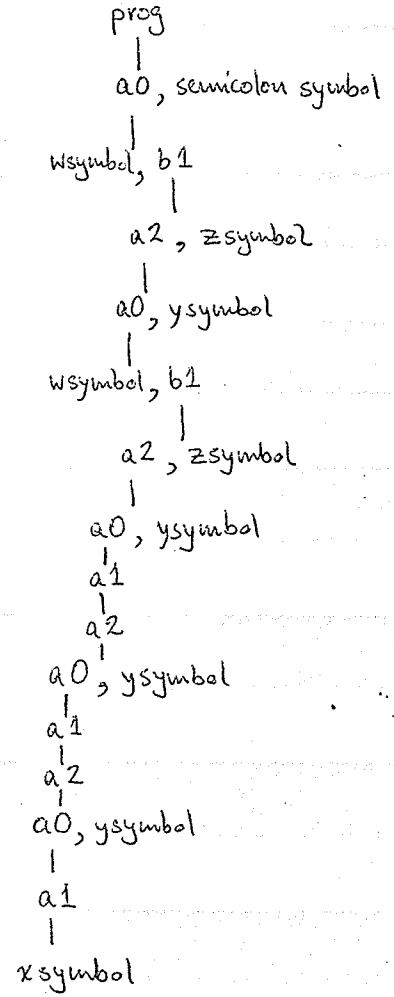


```
THE FOLLOWING ARRAY BOUNDS HAVE BEEN SPECIFIED OR ARE USED BY DEFAULT:
NAR=        50          NGOAL=        12          NRL=        10          NRS=        20          NSL=        12
NTL=         5          NSTRING=      10;


SYNTAX LIST
   NO.  BROTH   COMP    SON   CODE
     1      0      0      2      0
     2      0     12      3     12
     3      4      0      6      0
     4      0      5      1     10
     5      0      0     10      0
     6      7      0      8     11
     7      0      0      2     10
     8      0      0      3     11
     9      0      0      3     10
    10      0     11      8     12
    11      0      0      4     10
    12      0      0      5     10

TERMINAL LIST
     1 W
     2 X
     3 Y
     4 Z
     5 ;

INPUT STRING
WWXYYYZYZ;

RECURSION LIST
     1      3      2
     2      8      1
```

SENTENCE WAS FOUND
THE FINAL ANALYSIS RECORD FOLLOWS

NO.	POINT	SUB	SUB	PRED
1	1	0	25	0
2	2	1	4	0
3	4	2	0	0
4	5	2	26	3
5	10	4	23	0
6	8	5	8	0
7	4	6	0	0
8	5	6	22	7
9	10	8	21	0
10	8	9	11	0
11	3	10	12	0
12	6	11	20	0
13	8	12	14	0
14	3	13	15	0
15	6	14	19	0
16	8	15	17	0
17	3	16	18	0
18	7	17	0	0
19	9	15	0	16
20	9	12	0	13
21	9	9	0	10
22	11	8	0	9
23	9	5	0	6
24	11	4	0	5
25	12	1	0	21

ANALYSIS TREE

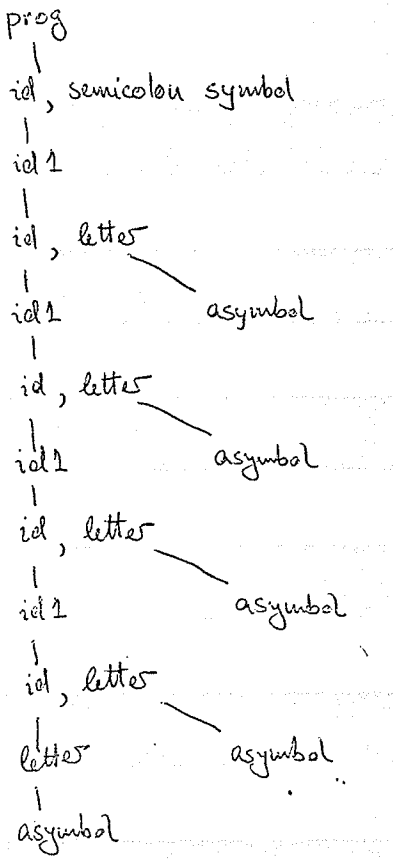


```
THE FOLLOWING ARRAY BOUNDS HAVE BEEN SPECIFIED OR ARE USED BY DEFAULT:
NAR=        50          NGOAL=        12          NPL=        10          NFS=        20          NSL=         8
NTL=         2          NSTRING=       6;


SYNTAX LIST
   NO.  BRCTH  COMP   SON  CODE
     1     0     0     2     0
     2     0     8     3    12
     3     0     0     4     0
     4     6     5     3    11
     5     6     0     7     0
     6     0     0     7     0
     7     0     0     1    10
     8     0     0     2    10

TERMINAL LIST
     1 A
     2 ;

INPUT STRING
AAAAA;

RECURSION LIST
     1     3     1
```

SENTENCE WAS FOUND
THE FINAL ANALYSIS RECORD FOLLOWS

NO.	POINT	SUB	SUB	PREDI
1	1	0	22	0
2	2	1	3	0
3	3	2	20	0
4	4	3	5	0
5	3	4	18	0
6	4	5	7	0
7	3	6	16	0
8	4	7	9	0
9	3	8	14	0
10	4	9	11	0
11	3	10	12	0
12	6	11	13	0
13	7	12	0	0
14	5	9	15	10
15	7	14	0	0
16	5	7	17	8
17	7	16	0	0
18	5	5	19	6
19	7	18	0	0
20	5	3	21	4
21	7	20	0	0
22	3	1	0	21

ANALYSIS TREE

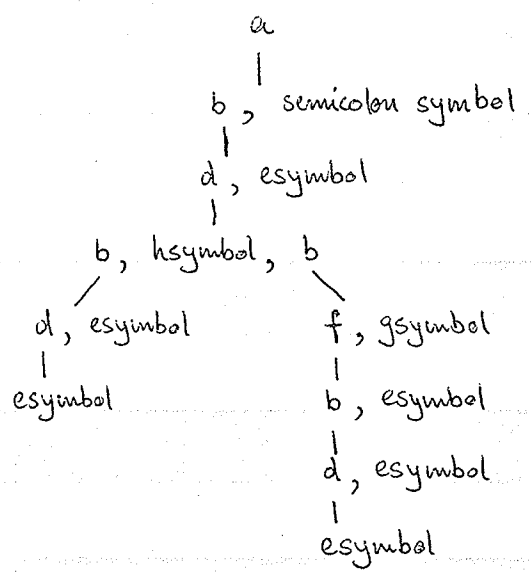


```
THE FOLLOWING ARRAY BOUNDS HAVE BEEN SPECIFIED OR ARE USED BY DEFAULT:
NAR=        50          NGOAL=        12          NPL=        10          NPS=        20          NSL=        13
NTL=         4          NSTRING=       9;


SYNTAX LIST
   NO. BROTH  COMP   SON  CODE
     1     0     0     2     0
     2     0     3     4    12
     3     0     0     1    10
     4     6     5     8     0
     5     6     0     2    10
     6     0     7    12     0
     7     0     0     3    10
     8    11     9     4    11
     9    11    10     4    10
    10    11     0     4    12
    11     0     0     2    10
    12     0    13     4    11
    13     0     0     2    10

TERMINAL LIST
     1 ;
     2 E
     3 G
     4 H

INPUT STRING
EEHEEEGE;

RECURSION LIST
     1     4     1
```

SENTENCE WAS FOUND
THE FINAL ANALYSIS RECORD FOLLOWS

NO.	POINT	SUB1	SUB1	PRED
1	1	0	18	0
2	2	1	17	0
3	4	2	9	0
4	8	3	7	0
5	4	4	6	0
6	11	5	0	0
7	5	4	0	5
8	9	3	0	4
9	10	3	16	8
10	6	9	15	0
11	12	10	14	0
12	4	11	13	0
13	11	12	0	0
14	5	11	0	12
15	13	10	0	11
16	7	9	0	10
17	5	2	0	3
18	3	1	0	2

ANALYSIS TREE



```
THE FOLLOWING ARRAY BOUNDS HAVE BEEN SPECIFIED OR ARE USED BY DEFAULT:
NAR=       50          NGOAL=       12          NFL=       10          NFS=       20          NSL=       12
NTL=        4          NSTRING=     8;


SYNTAX LIST
   NO. PROTH  COMP   SON  CODE
     1     0     0     2     0
     2     0     3     4    12
     3     0     0     4    10
     4     6     5     9     0
     5     6     0     1    10
     6     8     7    11     0
     7     8     0     1    10
     8     0     0     1    10
     9     0    10     4    11
    10     0     0     2    10
    11     0    12     4    11
    12     0     0     3    10

TERMINAL LIST
     1 A
     2 B
     3 C
     4 ;

INPUT STRING
ACABABA;

RECURSION LIST
     1     4     1
```

SENTENCE WAS FOUND
THE FINAL ANALYSIS RECORD FOLLOWS

NO.	POINT	SUB	SUB	PRED
1	1	0	16	0
2	2	1	15	0
3	4	2	14	0
4	9	3	13	0
5	4	4	12	0
6	9	5	11	0
7	6	6	10	0
8	11	7	9	0
9	8	9	0	0
10	12	7	0	8
11	7	6	0	7
12	10	5	0	6
13	5	4	0	5
14	10	3	0	4
15	5	2	0	3
16	3	1	0	2

ANALYSIS TREE

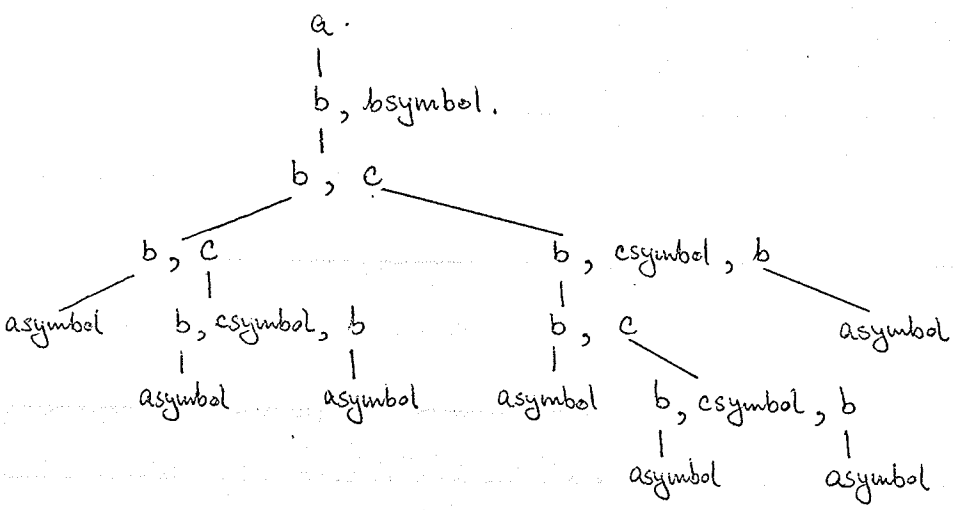
a
|
b, semicolon symbol
|
b1, asymbol
|
b, bsymbol
|
b1, asymbol
|
b, bsymbol
|
c, asymbol
|
b, csymbol
|
asymbol

```
THE FOLLOWING ARRAY BOUNDS HAVE BEEN SPECIFIED OR ARE USED BY DEFAULT:
NAR=        50          NGOAL=        12          NPL=        10          NPS=        20          NSL=         9
NTL=         3          NSTRING=      11;


SYNTAX LIST
   NO.  BROTH  COMP   SON  CODE
     1      0     0     2     0
     2      0     3     4    12
     3      0     0     2    10
     4      6     5     4    11
     5      6     0     7     0
     6      0     0     1    10
     7      0     8     4    12
     8      0     9     3    10
     9      0     0     4    12

TERMINAL LIST
     1 A
     2 B
     3 C

INPUT STRING
AACAAACACAB

RECURSION LIST
     1     4     1
```

THE FINAL ANALYSIS RECORD FOLLOWS

NO.	POINT	SUB1	SUB2	PRED
1	1	0	25	0
2	2	1	5	0
3	4	2	4	0
4	6	3	0	0
5	5	2	9	3
6	7	5	7	0
7	6	6	0	0
8	8	5	0	6
9	9	5	12	8
10	4	9	11	0
11	6	10	0	0
12	5	9	23	10
13	7	12	16	0
14	4	13	15	0
15	6	14	0	0
16	5	13	20	14
17	7	16	18	0
18	6	17	0	0
19	8	16	0	17
20	9	16	21	19
21	6	20	0	0
22	8	12	0	13
23	9	12	24	22
24	6	23	0	0
25	3	11	0	21

ANALYSIS TREE



THE FOLLOWING ARRAY BOUNDS HAVE BEEN SPECIFIED OR ARE USED BY DEFAULT:

NAR= 50 NGOAL= 12 NRL= 10 NRS= 20 NSL= 13
NTL= 7 NSTRING= 12;

SYNTAX LIST

NO.	BROTH	COMP	SON	CODE
1	0	0	2	0
2	0	3	4	12
3	0	0	2	10
4	6	5	7	12
5	6	0	3	10
6	0	0	1	10
7	9	8	4	11
8	9	0	7	10
9	0	10	11	0
10	0	0	4	10
11	13	12	7	11
12	13	0	5	10
13	0	0	6	10

TERMINAL LIST

1 A
2 B
3 C
4 D
5 E
6 F
7 G

INPUT STRING
FDEDCGEDCGCR

RECURSION LIST

1	4	1
2	7	2

THE FINAL ANALYSIS RECORD FOLLOWS

NO.	POINT	SUB	SUB	PRED
1	1	0	24	0
2	2	1	23	0
3	4	2	22	0
4	7	3	21	0
5	4	4	20	0
6	9	5	19	0
7	11	6	18	0
8	7	7	17	0
9	4	8	16	0
10	9	9	15	0
11	11	10	14	0
12	9	11	13	0
13	13	12	0	0
14	10	11	0	12
15	12	10	0	11
16	10	9	0	10
17	5	8	0	9
18	8	7	0	8
19	12	6	0	7
20	10	5	0	6
21	5	4	0	5
22	8	3	0	4
23	5	2	0	3
24	3	1	0	2

ANALYSIS TREE

a
|
b, bsymbol
|
c, csymbol
|
b, gsymbol
|
c, csymbol
|
d, dsymbol
|
c, esymbol
|
b, gsymbol
|
c, csymbol
|
d, dsymbol
|
c, esymbol
|
d, dsymbol
|
fsymbol

REFERENCES

1. N.Wirth and H. Weber: "EULER: A Generalization of ALGOL and its Formal Definition: Part I", CACM, Vol. 9/ No. 1/ January 1966

2. A. Learner and A.L. Lin: "A note on transforming context-free grammars to Wirth-Weber precedence form", The Computer Journal, Vol. 13/ No. 2/ May 1970

3. R.W. Floyd: "The Syntax of Programming Languages- A Survey", IEEE Transactions on Electronic Computers, Vol. EC-13/ August 1964

4. A. van Wijngaarden (Ed.) et al: "Report on the Algorithmic Language ALGOL 68", Mathematisch Centrum/ October 1969

5. B. Wiebe: "Two-level Syntax Analysis", forthcoming Thesis/ University of Manitoba

6. J. Feldman and D. Gries: "Translator Writing Systems", CACM, Vol. 11/No. 2/ February 1968

7. J.E. Hopcroft and J.D. Ullman: "Formal Languages and their Relation to Automata",

Addison-Wesley Publishing Company, 1969

8. D.J. Cohen and C.C. Gotlieb: "A List Structure Form of Grammars for Syntactic Analysis", Computing Surveys, Vol. 2/ No. 1/ March 1970

9. F.L. De Remer: "Practical Translators for LR(k) Languages", Ph. D. Thesis, M.I.T. October 1969

10. J. Earley: "An Efficient Context-free Parsing Algorithm", CACM, Vol. 13/ No. 2/ February 1970